L14: Entry 14 of 16 File: USPT Jun 4, 1996

DOCUMENT-IDENTIFIER: US 5524205 A
TITLE: Methods and apparatus for optimizing undo log usage

Abstract Text (1):
Each node in a data processing system contains at least one undo buffer and one least one redo buffer for insuring that any changes made to a section of a non-volatile storage medium, such as a disk, can be removed, if a transaction has not been committed, or can be recreated if the transaction has not been committed. The undo buffers each correspond to a different uncommitted transaction. The redo buffer contains the changes made to a copy of the section which is maintained in the memory.

Brief Summary Text (3):
The present invention relates generally to the field of recovery from crashes in shared disk systems, and in particular, to the use of logs in such recovery.

Brief Summary Text (4):
All computer systems may lose data if the computer crashes. Some systems, like data base systems, are particularly susceptible to possible loss of data from system failure or crash because those systems transfer great amounts of data back and forth between disks and processor memory.

Brief Summary Text (7):
The type of recovery needed depends, of course, on the reason for the loss of data. If a computer system crashes, the recovery needs to enable the restoration of the persistent storage, e.g. disks, of the computer system to a state consistent with that produced by the last committed transactions. If the persistent storage crashes (called a media failure), the recovery needs to recreate the data stored onto the disk.

Brief Summary Text (9):
Logs are difficult to manage, however, in system configurations where a number of computer systems, called "nodes," access a collection of shared disks. This type of configuration is called a "cluster" or a "shared disk" system. A system that allows any nodes in such a system to access any of the data is called a "data sharing" system.

Brief Summary Text (11):
In partitioned systems, as in single node or centralized systems, each portion of data can reside in the local memory of at most one node. Further, both partitioned systems and centralized systems need only record actions on a single log. Just as importantly, data recovery can proceed based solely on the contents of one log.

Brief Summary Text (12):
Distributed data shipping systems, on the other hand, are decentralized so the same data can reside in the local memories of multiple nodes and be updated from these nodes. This results in multiple nodes logging actions for the same data.

Brief Summary Text (15):
These difficulties are important to address because data sharing systems are often preferable to partitioned systems. For example, data sharing systems are important for workstations and engineering design applications because data sharing systems allow the workstations to cache data for extended periods which permits high performance local processing of the data. Furthermore, data sharing systems are inherently fault-tolerant and achieve load balancing because a multiplicity of nodes can access the data simultaneously, manage some local data themselves, and share other data with other host computers and workstations.

Brief Summary Text (18):
Another object of this invention is to minimize the information which must be stored to undo transactions in case of crashes or failures.

Brief Summary Text (21):
Specifically, in a data processing system including a plurality of nodes and a non-volatile storage medium divided into sections, the plurality of nodes making changes to the sections by way of transactions, each transaction comprising a series of changes made to at least one section by at least one node, and each transaction being committed if a record of the changes effected by that transaction as well as an indication of the completion of that transaction are reliably stored on the storage medium, and otherwise being uncommitted, a first one of the plurality of nodes comprises several elements. They include: a memory for holding a copy of at least one section; processing means, coupled to the memory, for making changes to the copy of the at least one section in the memory; at least one undo buffer containing a sequential list of the changes made by the processing means to the copy of the at least one section in the memory, each undo buffer corresponding to changes made by the first node to a different uncommitted transaction; a redo buffer containing a sequential list of changes made by the processing means in the corresponding node to the copy of the at least one section in memory; storing means, coupled to the memory, for storing the copy of the at least one section back into the storage medium; and log management means, coupled to the undo buffers and to the redo buffer, for selectively storing the portions of the undo buffers and redo buffer to the storage medium to ensure that the effects of all changes of uncommitted transactions can be removed and the effects of all changes of committed transactions can be recreated.

Detailed Description Text (4):
System 100 is an example of a storage system which can be used to implement the present invention. System 100 includes several nodes 110, 120, and 130, all accessing a shared disk system 140. Each of the nodes 110, 120, and 130 includes a processor 113, 123, and 133, respectively, to execute the storage and recovery routines described below. Nodes 110, 120, and 130 also each include a memory 118, 128, and 138, respectively, to provide at least two functions. One of the functions is to act as a local memory for the corresponding processor, and the other function is to hold the data being exchanged with disk system 140. The portions of memory that are used for data exchange are called caches. Caches are generally volatile system storage.

Detailed Description Text (6):
In addition, the persistent storage used to implement this invention is not limited to the architecture shown in FIG. 1. For example, the persistent storage could include several disks each coupled to a different node, with the nodes connected in some type of network.

Detailed Description Text (7):
Another part of persistent storage is a backup tape system 150 which is referred to as "archive storage." Archive storage is a term used generally to refer to the system storage used for information that permits reconstruction of the contents of persistent storage should the data in the persistent storage become unreadable. For example, should shared disk system 140 have a media failure, tape system 150 could be used to restore disk system 140. Archive storage frequently includes a magnetic tape system, but it could also include magnetic or optical disk systems as well.

Detailed Description Text (8):
Data in system 100 is usually stored in blocks, which are the recoverable objects of the system. In general, blocks can be operated upon only when they are in the cache of some node.

Detailed Description Text (11):
As explained above, most data base systems use logs for recovery purposes. The logs are generally stored in persistent storage. When a node is updating persistent storage, the node stores the log records describing the updates in a buffer in the node's cache.

Detailed Description Text (12):
The preferred implementation of the present invention envisions three types of logs in persistent storage, but only two types of buffers in each node's cache. The logs are redo logs, or RLOGs, undo logs, or ULOGs, and archive logs, or ALOGs. The buffers are the redo buffers and

the undo buffers.

Detailed Description Text (23):
One goal of this invention is to allow each node to manage its recovery as independently of the other nodes as possible. To do this, a separate RLOG is associated with each node. The association of an RLOG with a node in the preferred implementation involves use of a different RLOG for each node. Alternatively, the nodes can share RLOGs or each node can have multiple RLOGs. If an RLOG is private to a node, however, no synchronization involving messages is needed to coordinate the use of the RLOG with other RLOGs and nodes.

Detailed Description Text (41):
In order to update the DSI and prepare for the next operation, it is also necessary to be able to determine the ASI for a block after applying the log record. It is useful to be able to derive the ASI from the log record, such as from the BSI, so the ASI need not be stored in log records, although the ASI can indeed be stored. The derivation must be one, however, that can be used during recovery as well as during normal operation. Preferably, the SIs are in a known sequence, such as the monotonically increasing set of integers beginning with zero. In this technique, the ASI is always one greater than the BSI.

Detailed Description Text (43):
If the WAL protocol is not followed, and a block were to be written to persistent storage prior to the log record for the last update for the block, recovery could not occur under certain conditions. For example, an update at one node may cause a block containing uncommitted updates to be written to persistent storage. If the last update to that block has not been stored to the node's RLOG, and another transaction on a second node further updates the block and commits, the DSI for the block will be incremented. At the moment of commit for that second transaction, the logged actions for these other transactions are forced to the RLOG for the second node. Because the second update was generated by a different node, however, the writing of the log records for the second transaction does not assure that the log record for the uncommitted transaction on the original node is written.

Detailed Description Text (44):
If the original node crashes and the log record for the uncommitted transaction is never written to the RLOG, a gap is created in the ASI-BSI sequencing for the block. Should the block in persistent storage ever become unavailable, for example because of disk failure, recovery would fail because the ALOG merge, as explained below, requires a known and gapless sequence of SIs.

Detailed Description Text (45):
Thus, the WAL protocol is a necessary condition for an unbroken sequence of logged actions. It is also a sufficient condition with respect to block updates. When a block moves from one node's cache to another, the WAL protocol forces the RLOG records for all prior updates to the blocks to be changed by the committing transaction. "Forcing" means ensuring that the records in a nodes cache or buffer are stably stored in persistent storage.

Detailed Description Text (46):
By writing to persistent storage, the WAL protocol forces the writing of all records in the original node's RLOG up through the log record for the last update to the current block.

Detailed Description Text (52):
Of course, to make this procedure operate correctly, blocks containing space management information must be periodically written to persistent storage, and one node must not be allowed to reallocate blocks freed by another node until the freed blocks' existence is made known to it via this bookkeeping. Thus, maintaining initial SI's for freed blocks does not cause additional reading or writing of the free space bookkeeping information.

Detailed Description Text (58):
For purposes of recovery, there are three kinds of blocks. A version of a block is "current" if all updates that have been performed on the block are reflected in the version. A block having a current version after a failure needs no redo recovery. When dealing with unpredictable system failures, however, one cannot ensure that all blocks are current without always "writing-thru" the cache to persistent storage whenever an update occurs. This is expensive and is rarely done.

Detailed Description Text (59):
A version of a block is "one-log" if only one node's log has updates that have not yet been applied to the block. When a failure occurs, at most one node need be involved in recovery. This is desirable because it avoids potentially extensive coordination during recovery, as well as additional implementation cost.

Detailed Description Text (60):
A version of a block is "N-log" if more than one node's log can have updates that have not yet been applied to it. Recovery is generally more difficult for N-log blocks than one-log blocks, but it is impractical when providing media recovery to ensure that blocks are always one-log because this would involve writing a block to archive storage every time the block changes nodes.

Detailed Description Text (62):
Without care, some blocks will be N-log at the time of a system crash (as opposed to a media failure). The preferred implementation of this invention, however, guarantees that all blocks will be one-log blocks for system crash recovery. This is advantageous because N-log blocks can require complex coordination between nodes for their recovery. Although such coordination is possible since the updates were originally sequenced during normal system operation using distributed concurrency control, such concurrency control requires overhead which should be avoided during recovery.

Detailed Description Text (64):
If this rule is followed, a requesting node always gets a clean block when the block enters the new node's cache. Furthermore, during recovery, only the records on the log of the last node to change the block need be applied to the block. All other actions of other nodes have already been captured in the state of the block in persistent storage. Thus, all blocks will be one-log for redo recovery, so redo recovery will not require distributed concurrency control.

Detailed Description Text (65):
Following this technique does not mean that multiple logs will never contain records for a block. This technique merely ensures that only one node's records are applicable to the version of the block in persistent storage.

Detailed Description Text (66):
Furthermore, although one-log redo recovery is being assumed for system crashes, in order to perform media recovery, redo actions on multiple logs may have to be applied to avoid writing each block to archive storage every time the block moves between caches. Hence, it is still necessary in certain circumstances to order the logged actions across all the logs to provide recovery for N-log blocks. This can be accomplished, however, because of the sequential SIs.

Detailed Description Text (67):
FIG. 6 shows a flow diagram 600 of the basic steps for a redo operation using the RLOG and the SIs described above. The redo operation represented by flow diagram 600 would be performed by a single node using a single RLOG record applied to a single block.

Detailed Description Text (70):
The redo operation described with regard to FIG. 6 can be used in recovering from system crashes. An example of a procedure of crash recovery is shown by the flow diagram 700 in FIG. 7. A single node can execute this crash recovery procedure independently of other nodes.

Detailed Description Text (71):
The first step would be for the node to read the first RLOG record indicated by the most recent checkpoint (step 710). The checkpoint, as described below, indicates the point in the RLOG which contains the record corresponding to the oldest update that needs to be applied.

Detailed Description Text (74):
If the SI associated with a log record is a monotonically increasing ASI, the test of whether a log record applies to a block in some state is whether this ASI is the first one greater than the block's DSI. This is sufficient only for one-log recovery, however, because in that case only one log will have records with ASIs that are greater than the DSI in the block.

Detailed Description Text (79):
Media recovery is N-log because it involves restoring blocks from archive storage and, as explained above, blocks are not written to archive storage every time they move between caches. Thus the technique of writing blocks to storage to avoid N-log recovery for system crashes cannot be used for media recovery.

Detailed Description Text (83):
Beginning with any ALOG, the first log record is accessed (step 810). The Block ID and BSI are then extracted from that record (step 820). Next, the block identified by the Block ID is fetched (step 830).

Detailed Description Text (96):
Safe point determination is important to determine how much of the RLOG needs to be scanned in order to perform redo recovery. The starting point in the RLOG for this redo scan is called the "safe point." The safe point is "safe" in two senses. First, redo recovery can safely ignore records that precede the safe point since those records are all already included in the versions of blocks in persistent storage. Second, the "ignored" records can be truncated from the RLOG because they are no longer needed.

Detailed Description Text (101):
Another entry in Dirty Blocks table 900 is the LastLSN entry 950. The value for this entry is, for each block, the LSNs of the RLOG and ULOG records that describe the last update to the block. LSNs are used rather than DSIs because it is necessary to determine locations in logs.

Detailed Description Text (104):
The earliest LSN for all blocks in a node's cache is the safe point for the redo scan in the local RLOG. Redo recovery is started by reading the local RLOG from the safe point forward and redoing the actions in subsequent records. All blocks needing redo have all actions needing to be redone encountered during this scan.

Detailed Description Text (105):
As explained above, the one-log assumption makes it possible to manage each RLOG in isolation. A node need only deal with its own RLOG, thus one node's actions will never be the reason for a block being dirty in some other node's cache. Hence, it is sufficient to keep a simple recovery LSN (one that does not name the RLOG) associated with each block, where it is understood that the recovery LSN identifies a record in the local RLOG.

Detailed Description Text (112):
To truncate an ALOG, blocks on persistent storages are first backed up to archive storage. When this is complete, an archive checkpoint record is written to an agreed upon location, e.g., in archive storage, to identify the RLOG checkpoints that were current when determination of the archive checkpoint began.

Detailed Description Text (114):
Checkpoints are written to the RLOG. To find the last checkpoint written to the RLOG, its location is written to the corresponding node's persistent storage in an area of global information for the node. The most recent checkpoint information is typically the first information accessed during recovery. Alternatively, one can search the tail of the RLOG for the last checkpoint.

Detailed Description Text (117):
The system exercises control over the RLOG by writing blocks back to their locations in persistent storage. In fact, this writing of blocks is sometimes considered part of the checkpoint. Blocks may also be written to persistent storage that have recovery LSNs that are older, i.e., further back in the RLOG. This moves the safe point for the RLOG closer to the tail of the log. Log records whose operations are included in the newly-written block are no longer needed for redo recovery, and hence can be truncated.

Detailed Description Text (125):
For N-log undo, multiple nodes can have uncommitted data in a block simultaneously. A system crash would require these transactions to all be undone, which may require, for example, locking during undo recovery to coordinate block accesses.

Detailed Description Text (126):
To ensure that all blocks will be one-log with respect to undo recovery, no block containing uncommitted data from one node is ever permitted to be updated by a second node. This can be achieved through a lock granularity that is no smaller than a block. A requesting node will then receive a block in which no undo processing by another node is ever required. Therefore, for example, if a transaction from another node had updated a block and then aborts, the effect of that transaction has already been undone.

Detailed Description Text (154):
LSN 1170, which need not be stored explicitly because it may be identified by its location in the RLOG, identifies this CLR uniquely on the RLOG. The LSN is used to control the redo scan and checkpointing of the RLOG.

Detailed Description Text (161):
STATE attribute 1220 indicates whether an active transaction is "prepared" as part of a two-phase commit. A two-phase commit is used when multiple nodes take part in a transaction. To commit such a transaction, all the nodes must first prepare the transaction (phase 1) before they can commit it (phase 2). The preparation is done to avoid partial commits which would occur if one node commits, but another aborts. A prepared transaction needs to be retained in the Active Transactions table 1200 because it may need to be rolled back. Unlike a non-prepared transaction, a prepared transaction should not be automatically aborted.

Detailed Description Text (162):
ULOGloc attribute 1230 indicates the location of the transaction-specific ULOG. This attribute need only be present should there be no other way to find the ULOG. For example, the TID 1210 might provide a substitute way of finding the ULOG for the transaction.

Detailed Description Text (173):
The goal is to support N-log undo where several nodes may undo transactions on a single block as a result of a system crash. Hence, the progress of undo operations performed by each node must be stably recorded. This is what CLRs accomplish in the NO-FORCE case. Without CLRs, some other technique is required.

Detailed Description Text (198):
Some blocks may be read by several nodes to determine whether they need to be involved in local redo, but only one of the nodes will actually perform redo for a block. This can, however, be almost completely avoided by writing block-write records to the RLOG. Because block-write records need not be forced, a block will occasionally be read from persistent storage when this is not necessary. The penalty for such a read is small, however.

Detailed Description Text (199):
A one-log version of every block exists in persistent storage, so only one node can have records in its log that have a BSI equal to the DSI of the block. This node is the one that will independently perform redo processing on the block. Hence, redo can be done in parallel by the separate nodes of the system, each with its own RLOG. No concurrency control is needed here.

Detailed Description Text (200):
The redo phase reconstructs the state of the node's cache by accessing the dirty blocks needing redo and posting the changes as indicated in the RLOG records. The resulting cache contains the dirty blocks in their states as of the time of the crash. Blocks that were subject to redo have been locked. The resulting Dirty Blocks table 900 and Active Transactions table 1200 are similarly reconstructed. Blocks that were subject to redo have been locked.

Detailed Description Text (204):
In the application of an RLOG record to a block, the block's DSI is updated to the ASI for the redone action. The node requests an appropriate lock on the block when an RLOG action is applied. Redo need not wait for the lock to be granted, since no other node will request a lock. The requested locks must, however, be granted prior to the start of undo. This is the way concurrency control is initialized for the undo phase.

Detailed Description Text (209):
Undo recovery is N-log. Hence, the undo recovery phase needs concurrency control in the same

way that it is needed during transaction rollback. Multiple nodes may need to undo changes to the same block. Normal data base activity can resume once the undo phase begins, however, just as normal activity can proceed concurrently with transaction abort. All the appropriate locking is in place to permit this. This is ensured by not starting the undo phase until all nodes have completed the redo phase. Hence, all locks requested by any node during redo are held by the appropriate node prior to undo beginning.

Detailed Description Text (214):
It will be apparent to persons of ordinary skill in the art that modifications and variations can be made without departing from the spirit and scope of this invention. For example, the architecture shown in FIG. 1 may be different, and the number of undo and redo logs assigned to each node can vary. The present invention covers such modifications and variations which come within the scope of the appended claims and their equivalents.

Previous Doc Next Doc Go to Doc#

L4: Entry 4 of 4 File: USPT Oct 24, 2000

DOCUMENT-IDENTIFIER: US 6138125 A
TITLE: Block coding method and system for failure recovery in disk arrays

Abstract Text (1):
A method, system, and data structure for encoding a block of data with redundancy information and for correction of erasure type errors in the block using the redundancy data. In particular, the invention is particularly applicable to disk array storage subsystems which are capable of recovering from total or partial failures of one or two disks in the disk array. Still more specifically, the invention is applicable to RAID level 6 storage devices. A given data block of data is translated into a code block of n.sup.2 elements including 2n XOR parity elements for redundancy. Each code block is manipulated as a square matrix, of n.sup.2 elements with parity elements along the major diagonals of the matrix and data elements in the remainder of the matrix. Each parity element is a dependent variable whose value is the XOR sum of the (n-2) data elements in a minor diagonal which intersects it. If the elements in any one or two columns or one or two rows are erased, their values can be generated from the other elements in the matrix. The invention therefore allows for recovery from data loss resulting from complete failure of any one or two disks in the disk array. Further, since the invention recovers all erased elements in any one or two rows, it allows recovery from data loss resulting from correlated partial failure of all disks in the disk array. Still further, the invention allows recovery from many uncorrelated failure patterns in the storage domain of disk drives in a disk array.

Brief Summary Text (6):
However, multiplying the number of disk drives used to store data increases the probability of a failure causing loss of data. The mean time between failure of multiple disk drives storing data is less than the mean time between failure of a single disk drive storing the same data. Storage arrays therefore provide for additional (overhead) storage to provide redundancy information used to recover data lost due to failure of other disk drives. Such loss of data is often referred to as "erasure" of the data. The redundancy information is used in general for two purposes. First, the redundancy information is used to restore data to a failed disk drive after it is repaired or replaced. Second, the redundancy information is used to allow continued operation of the storage array system while the failed disk drive is undergoing repair or replacement. In other words, lost data on the failed disk drive may be regenerated in real time by the storage array system using the redundancy information.

Brief Summary Text (10):
XOR parity as provided in RAID 2-5 guards against the loss of data from failure of a single disk drive in the array. When a single disk drive fails, the data lost on that disk drive is reconstructed by performing an XOR of the related blocks of the data in a corresponding stripe on the remaining operational disk drives. As noted above, the lost data may be reconstructed in real time to continue operation of the array despite the loss of a single disk and may be reconstructed at the time of replacement of the failed disk drive with a replacement or repaired disk.

Detailed Description Text (6):
The various RAID levels are distinguished by the manner in which array controller 102 logically subdivides or partitions the disks 110(*) in disk array 108. RAID Level 6 systems provide protection from failure of two disks in an array. RAID level 6 requires two independent redundancy computations be used for each portion of protected data. Two such independent methods assure recovery of data even in the event of failure of two disk drives 110(*) in disk array 108.

Current US Original Classification (1):
707/202

Current US Cross Reference Classification (1):
707/205

Previous Doc Next Doc Go to Doc#

L13: Entry 3 of 4 File: USPT Aug 13, 2002

DOCUMENT-IDENTIFIER: US 6434555 B1
TITLE: Method for transaction recovery in three-tier applications

Drawing Description Text (3):
FIG. 2 is a timeline illustrating the operation of the invention in the case of a single failure-free database;

Detailed Description Text (17):
As a result, the manipulation 616 by the request processor 608 returns an exception 620 that is passed on to the client 600. The transaction coordinator 610 also detects 622 the database failure 618 and begins the recovery process by sending itself the rebuild c-set command 624. In order to rebuild the c-set, the transaction coordinator 610 sends a committed query 626 to the databases DB.sub.1 604 through DB.sub.n 606. In return, the databases DB.sub.1 604 through DB.sub.n 606 reply 628 with information to compile a list of committed transactions. To be safe, the transaction coordinator 610 sends forget messages 630 to the databases DB.sub.1 604 through DB.sub.n 606 to confirm that all the expired transactions in the committed transaction list are forgotten.

CLAIMS:

7. A method of transaction recovery from a failure in a three-tier transaction processing system having a client machine, an application server including a request processor and a transaction coordinator, and a database machine using a database recovery log, comprising the steps of: providing an interface between the client machine, the application server, and the database machine with the database recovery log; receiving a request from the client machine by the request processor that a transaction be processed; starting data manipulation of the database machine by the request processor; manipulating data in the database machine by the request processor; ending data manipulation in the database machine by the request processor; receiving a terminate command by the transaction coordinator from the request processor; sending a commit command to commit the transaction to the database machine from the transaction coordinator; logging the commit of the transaction in the database recovery log; providing an acknowledgement that the transaction has been committed to the transaction coordinator by the database machine; providing a reply to the client machine from the application server acknowledging the completion of the transaction termination process; receiving a forget command by the request processor from the client machine; receiving a forget command by the transaction coordinator from the request processor; and receiving a forget command by the database machine from the transaction coordinator.

10. A method of transaction recovery from a failure in a three-tier transaction processing system having a client machine, an application server, and a plurality of database machines using database recovery logs, comprising the steps of: receiving a request from the client machine by the request processor that a transaction be processed; starting data manipulation of the plurality of database machines by the request processor; manipulating data in the plurality of database machines by the request processor; ending data manipulation in the plurality of database machines by the request processor; receiving a terminate command by the transaction coordinator from the request processor; commanding the transaction to be prepared by the plurality of database machines from the application server; logging the transaction being prepared by the plurality of databases using the database recovery logs; providing an acknowledgement to the application server by the plurality of database machines that the transaction has been prepared; commanding the transaction to be committed by the plurality of database machines from the application server; logging the commit of the transaction in the database recovery logs; providing an acknowledgement to the transaction coordinator by the plurality of database machines that the transaction has been committed; sending a reply to the

request processor from the transaction coordinator acknowledging the completion of the transaction; providing a reply to the client machine from the request processor acknowledging the completion of the transaction; receiving of a forget command by the request processor from the client machine; receiving of a forget command by the transaction coordinator from the request processor; receiving of a forget command by the database machine from the transaction coordinator.

Previous Doc Next Doc Go to Doc#

L13: Entry 4 of 4 File: USPT Mar 20, 1984

DOCUMENT-IDENTIFIER: US 4438494 A
TITLE: Apparatus of fault-handling in a multiprocessing system

Detailed Description Text (97):
The MACD Bus confinement area is shown within the dotted lines of FIG. 5. The MACD bus confinement area is protected by three independent mechanisms. The MACD and CTL lines are protected by two parity bits (CHKO, 1). The bus protocol is such that only one device drives the data and control lines at any point in time (i.e., no handshaking). Two parity bits are used in an interleaved fashion to gain protection against certain classes of multiple failures. The parity mechanism can be used to protect the TTL buffers at the bus interfaces.

Detailed Description Text (127):
The error-reporting network must be functional for the machine to handle error conditions correctly. Although certain error conditions can be tolerated, there are single failures in the error-report network which will prevent the correct reporting of errors in the system. The report line exercises are designed to confirm that the report lines are all operating correctly.

Detailed Description Text (147):
Whenever an error is detected in the central system (the central system does not include IP:PS interface), the detecting node crosspoint switch reports the error occurrence to all the other nodes in the system. The error report describes the type of error which occurred and the ID of the node which detected the error. An error-report message is broadcast over the reporting network matrix (MERL and BERL) which follows the topology of the bus matrix (ACD and MACD). This reporting network has no single-point dependencies (assuming two or more processor modules), and errors are localized to either a bus or module confinement area. This message is uniformly logged throughout the central system. Each node then begins the recovery process. During recovery, higher authorities (processor microcode) may be informed of the reported error.

Detailed Description Text (178):
The error-report logs are used independently by both hardware and software. For proper hardware recovery, the log must always hold the information from the most recent error-report message. From a software point of view, the log must satisfy two needs: (1) provide consistent error log data to the requestor, and (2) uniquely identify error reports so that software can distinguish between the single occurence of an error and repeated occurrences of the same error.

Detailed Description Text (234):
The error-report lines allow the hardware system to report errors and begin recovery at the hardware level. If the fault was permanent, or if recovery was not successful, the hardware level must immediately inform system software of the problem. There are two signals used for this purpose:

Detailed Description Text (516):
Whenever an error is detected in the central system lines (133), the detecting node crosspoint switch reports the error occurrence to all the other nodes in the system. An error report generated by the generator (134) describes the type of error which occurred and the node ID (135) of the node which detected the error. An error-report message is broadcast over the reporting network matrix (MERL 106; BERL 108). This message is uniformly logged in the logger (138) of each crosspoint switch throughout the system, including the node that detected the error. Each node then begins the recovery process.

Previous Doc Next Doc Go to Doc#

L14: Entry 10 of 16 File: USPT Oct 12, 1999

DOCUMENT-IDENTIFIER: US 5966706 A
TITLE: Local logging in a distributed database management computer system

Abstract Text (1):
A distributed database management computer system includes a plurality of nodes and a plurality of database pages. When a first node in the computer system updates a first database page, the first node generates a log record. The first node determines whether it manages the first database page. If the first node determines that it manages the first database page, the first node writes the log record to a log storage local to the first node. However, if the first node determines that it does not manage the first database page, the first node then determines whether it includes a local log storage. If the first node includes a local log storage, the first node writes the log record to the local log storage, even if the first node does not manage the first database page. If the first node does not include a local log storage, the first node sends the log record to a second node managing the first database page.

Brief Summary Text (2):
The present invention is directed to a distributed database management computer system. More particularly, the present invention is directed to a distributed database management computer system which includes local logging of nodes and associated transaction recovery techniques.

Brief Summary Text (5):
Today, the major distributed database management computer systems are "client-server", "shared nothing" and "shared disks" architectures. Most of these architectures use logging for recovery. In a client-server system, both the database and the log are stored with the server and all log records generated by the clients are sent to the server. In a shared nothing system, the database is partitioned among several nodes and each node has its own log file. Each database partition is accessed only by the owning node and a distributed commit protocol is required for committing transactions that access multiple partitions. In a shared disks system, the database is shared among the different nodes. Some shared disks systems use only one log file and require system wide synchronization for appending log records to the log. An example of this known type of system is disclosed in T. Rengarajan et al., High Availability Mechanisms of VAX DBMS Software, Digital Technical Journal 8, pages 88-98, February 1989. Some other shared disks systems use a log file per node. An example of this known type of system is disclosed in D. Lomet, Recovery for Shared Disk Systems Using Multiple Redo Logs, Technical Report CLR 90/4, Digital Equipment Corp., Cambridge Research Lab, Cambridge, Mass., October 1990. However, these systems either force pages to disks when these pages are exchanged between two nodes or they merge the log files during a node crash.

Brief Summary Text (6):
FIG. 1 is a flowchart illustrating the steps performed by most known systems when a database page "P" is updated by an application running on a node "N." These steps are performed in most known client-server database systems that implement logging, as well as in any other known distributed database management computer systems with multiple nodes N.

Brief Summary Text (7):
In step 50 of FIG. 1, the database page P is updated by node N and stored in N's cache. In step 52, a log record of the update is generated by node N. In step 54, node N determines if page P is managed by node N. If it is, then in step 56, node N writes the log record to a local log disk. However, if at step 54 node N determines that page P is managed by another node, then at step 58 node N sends the log record to the node or server that manages page P.

Brief Summary Text (8):
As shown in the FIG. 1, in most known distributed database management computer systems, log

records are always stored local to the node that is managing the database page that created the log record.

Brief Summary Text (18):
E. Rahm, Recovery Concepts for Data Sharing Systems, Proc. 21st Int. Conf. on Fault-Tolerant Computing, Montreal, June 1991 discloses logging and recovery protocols for a shared disks architecture employing the "primary copy authority" (PCA) locking protocol. Under the PCA locking protocol, the entire lock space is divided among the participating nodes and a lock request for a given item is forwarded to the node responsible for that item. PCA supports only physical logging, not logical logging. PCA employs the no-steal buffer management policy in which only pages containing committed data are written to disk. This is an inflexible and expensive policy.

Brief Summary Text (19):
PCA allows pages to be modified by many systems before they are written to disk. However, commit processing involves the sending of each updated page to the node that holds the PCA for that page. Furthermore, double logging is required for every page that is modified by a node other than the PCA node. During normal transaction processing the modifying node writes log records in its own log and at transaction commit it sends all the log records written for remote pages to the PCA nodes responsible for those pages.

Brief Summary Text (20):
C. Mohan et al., Recovery and Coherency Control Protocols for Fast Intersystem Page Transfer and Fine-Granularity Locking in a Shared Disks Transaction Environment, Proceedings of the Seventeenth International Conference on Very Large Databases, Barcelona, Spain, pages 193-207, September 1991 ("Mohan et al.") discloses four different recovery schemes for a shared disk architecture. The schemes are designed to exploit the fast internode communication paths usually found in tightly coupled data sharing architectures, and they use fine granularity locking. However, the schemes assume that the clocks of all the nodes are perfectly synchronized. In addition, the simple and medium schemes disclosed in Mohan et al. force pages to disk when they are exchanged between nodes, and all of the disclosed schemes require merging of the local logs. In the fast and super-fast schemes disclosed in Mohan et al., private logs have to be merged even in the case where only a single node crashes.

Brief Summary Text (21):
D. Lomet, Recovery for Shared Disk Systems Using Multiple Redo Logs, Technical Report CLR 90/4, Digital Equipment Corp., Cambridge Research Lab, Cambridge, Mass., October 1990 discloses a shared data/private log recovery algorithm. However, the recovery algorithm requires a seamless ordering of page sequence numbers (PSNs) and associates for each database page extra information with the space management subsystem. In addition, the disclosed recovery algorithm forces modified pages to disk before they are replaced from a node's cache.

Brief Summary Text (22):
T. Rengarajan et al., High Availability Mechanisms of VAX DBMS Software, Digital Technical Journal 8, pages 88-98, February 1989 discloses a Rdb/VMS that is a data sharing database system executing on a VAX cluster. Earlier versions of Rdb/VMS employed an undo/no-redo recovery protocol that required, at transaction commit, the forcing to disk of all the pages updated by the committing transaction. More recent versions, disclosed in D. Lomet et al., How the Rdb/VMS Data Sharing System Became Fast, Technical Report CRL 92/4, Digital Equipment Corporation Cambridge Research Lab, 1992, offer both an undo/no-redo and an undo/redo recovery scheme. In addition, a variation of the callback locking algorithm, referred to as "lock carryover", is used for reducing the number of messages sent across the nodes for locking purposes. However, Rdb/VMS does not allow multiple outstanding updates belonging to different nodes to be present on a database page. Thus, modified pages are forced to disk before they are shipped from one node to another.

Brief Summary Text (23):
In Rdb/VMS, each application process can take its own checkpoint after the completion of a particular transaction. The checkpointing process forces to disk all modified and committed database pages. In addition, Rdb/VMS uses only one global log file. Consequently, the common log becomes a bottleneck and a global lock must be acquired by each node that needs to append some log records at the end of the log.

Brief Summary Text (25):
M. Satyanarayanan et al., Coda: A Highly Available File System for a Distributed Workstation Environment, IEEE Transactions on Computers, 39(4), April 1990 ("Coda") discloses a distributed file system operating on a network of UNIX workstations. Coda is based on the Andrew File System and cache coherency is based on the callback locking algorithm. The granularity of caching is that of entire files and directories. Coda can handle server and network failures and support portable workstations by using client's disks for logging. This ability is based on the disconnected mode of operation that allows clients to continue accessing and modifying the cached data even when they are not connected to the network. All updates are logged and they are reintegrated to the systems on reconnection.

Brief Summary Text (26):
However, Coda does not support failure atomicity, and updates cannot be rolled back. In addition, Coda does not guarantee that the updates performed by a transaction survive various system failures and they are altered only when a later transaction modifies them. Coda only guarantees permanence conditionally; updates made by a transaction may change if a conflict is discovered at the time these updates are being reintegrated into the system.

Brief Summary Text (27):
Based on the foregoing, there is a need for a distributed database management system in which: (1) updated pages are not forced to disk at transaction commit time or when they are replaced from a node cache, (2) transaction rollback and node crash recovery are handled exclusively by the nodes, (3) local log files are never merged during the recovery process, (4) each node can take a checkpoint without synchronizing with the rest of the operational nodes, and (5) clocks do not have to be synchronized across the nodes and lock tables are not checkpointed.

Brief Summary Text (29):
The present invention is a distributed database management computer system that includes a plurality of nodes and a plurality of database pages. When a first node in the computer system updates a first database page, the first node generates a log record. The first node determines whether it manages the first database page. If the first node determines that it manages the first database page, the first node writes the log record to a log storage local to the first node. However, if the first node determines that it does not manage the first database page, the first node then determines whether it includes a local log storage. If the first node includes a local log storage, the first node writes the log record to the local log storage, even if the first node does not manage the first database page. If the first node does not include a local log storage, the first node sends the log record to a second node managing the first database page.

Drawing Description Text (2):
FIG. 1 is a flowchart illustrating the steps performed by prior art systems when a database page is updated by an application running on a node.

Drawing Description Text (4):
FIG. 3 is a flowchart illustrating an overview of the steps performed by the present invention when a database page is updated by an application running on node.

Detailed Description Text (3):
The system of FIG. 2 consists of several networked processing nodes 10, 20, 30, 40. Nodes 10, 20, 30, 40 can also be referred to as clients and servers in a client/server arrangement. Some nodes have databases attached to it. For example, node 10 has database 12 attached to it, and node 30 has database 32 attached to it. A node having databases attached to it is referred to as "owner node" with respect to the items stored in these databases. All owner nodes have local logs. Therefore, node 10 has local log 11 and node 30 has local log 31. Nodes that do not own any database, such as nodes 20 and 40, may or may not have local logs. As shown, node 20 has local log 21. Although nodes with no local logs may participate in a distributed computation, the present invention applies only to nodes that have local logs.

Detailed Description Text (4):
A user program running on node N accesses data items that are owned by either N or some other remote node. These data items are fetched in N's cache, i.e., a data shipping architecture is assumed. Log records for data updated by N are written to the local log file and transaction commitment is carried out by N without communication with the remote nodes. To accomplish this,

the present invention correctly handles transaction aborts and node crashes, while incurring minimal overhead during normal transaction processing. The present invention includes the following features:

•

Detailed Description Text (5):
Log records for updates to cached pages are written to the log file of each node.

Detailed Description Text (6):
Transaction rollback and node crash recovery are handled exclusively by each node.

Detailed Description Text (7):
Node log files are not merged at any time.

Detailed Description Text (8):
Node clocks do not have to be synchronized.

Detailed Description Text (9):
Nodes can take checkpoints independently of each other.

Detailed Description Text (10):
FIG. 3 is a flowchart illustrating an overview of the steps performed by the present invention when a database page P is updated by an application running on node N.

Detailed Description Text (11):
In step 60 of FIG. 3, the database page P is updated by node N and stored in N's cache. In step 62, a log record of the update is generated by node N. In step 64, node N determines if page P is managed by node N. If it is, then in step 68, node N writes the log record to a local log disk. However, if at step 64 node N determines that page P is managed by another node, then at step 66 node N determines whether it has a local log disk. If node N has a local log disk, then at step 68 the log record is stored in the local log disk. If at step 66 it is determined that node N does not have a local log disk, then at step 70 node N sends the log record to the node or server that is managing page P.

Detailed Description Text (12):
As shown in FIG. 3, in contrast to the steps performed by prior art systems, the present invention allows for local storage of log records at node N even if the page P for which the log record was created is not managed by node N. The following sections include additional details of the steps shown in FIG. 3, and details on how error recovery is accomplished with a system executing the steps shown in FIG. 3

Detailed Description Text (14):
In the exemplary distributed database management computer system shown in FIG. 2, transactions are executed in their entirety in the node where they are started. Data items referenced by a transaction are fetched from the owner node before they are accessed. The unit of internode transfer is assumed to be a database page. Each node 10, 20, 30, 40 has a buffer pool (node cache) where frequently accessed pages are cached to minimize disk I/O and communication with owner nodes. The buffer manager of each node follows the "steal" and "no-force" strategies. These strategies are known in the art and are disclosed, for example, in T. Haerder et al., Principles of Transaction Oriented Database Recovery--A Taxonomy, ACM Computing Survey, pages 289-317, December 1983, incorporated herein by reference. Pages containing uncommitted updates that are replaced from the local cache are either written in-place to disk or sent to the owner node, depending on whether they belong to the local database. Pages that were updated by a terminated transaction (committed or aborted) are not necessarily written to disk or sent to the owner node before the termination of the transaction.

Detailed Description Text (15):
Further, in the distributed database management computer system shown in FIG. 2, concurrency control is based on locking and the strict two-phase locking protocol is used. Each node 10, 20, 30, 40 has a lock manager that caches the acquired locks and forwards the lock requests for data items owned by another node to that node. Each node maintains both the cached pages and the cached locks across transaction boundaries. This is referred to as inter-transaction caching and is disclosed, for example, in K. Wilkinson et al., Maintaining Consistency of Client-cached Data, Proceedings of the Sixteenth International Conference on Very Large

Databases, Brisbane, pages 122-133, August 1990, incorporated herein by reference. The callback locking protocol, disclosed in J. H. Howard, Scale and Performance in a Distributed File System, ACM Transactions on Computer Systems, 6(1):51-81, February 1988, incorporated herein by reference, is used for cache consistency.

Detailed Description Text (16):
Further, in the distributed database management computer system shown in FIG. 2, both shared and exclusive locks are retained by the node after a transaction terminates (whether committing or rolling back). Cached locks that are called back in exclusive mode are released and exclusive locks that are called back in shared mode are demoted to shared. The granularity of both locking and callback is assumed to be at the level of a database page. Optionally, fine-granularity locking is supported.

Detailed Description Text (17):
Further, in the distributed database management computer system shown in FIG. 2, each database page consists of a header that among other information contains a page sequence number (PSN), which is incremented by one every time the page is updated. The owner node initializes the PSN value of a page when this page is allocated by following the known approach disclosed in C. Mohan et al., ARIES/CSA: A Method For Database Recovery in Client-Server Architectures, Proceedings of ACM-SIGMOD 1994 International Conference on Management of Data, Minneapolis, Minn., pages 55-66, May 1994, incorporated herein by reference (i.e., the PSN stored on the space allocation map containing information about the page in question is assigned to the PSN field of the page).

Detailed Description Text (18):
Further, the log of each node is used for logging transaction updates, rolling back aborted transactions, and recovering from crashes. Recovery is based on the write ahead log (WAL) protocol and the known ARIES redo-undo algorithm (hereinafter referred to as the "ARIES algorithm", or "ARIES"), disclosed in C. Mohan et al., ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks using Write-Ahead Logging, ACM Transactions on Database Systems, 17(1):94-162, March 1992, incorporated herein by reference, is employed. Log records are written to the local log before an updated page is replaced from the node cache and at transaction commit. Each node log manager associates with each log record a log sequence number (LSN) that corresponds to the address of the log record in the local log file. Log records describing an update on a page contain among other fields the page id and the PSN the page had just before it was updated.

Detailed Description Text (23):
In the present invention, it is assumed that each client in the system writes log records for updates to pages in its own log file. It is also assumed that the crashed client performs restart recovery. However, the present invention does not require that each client has a log file, nor does it require that the crashed client is the one that will recover from its failure. In particular, clients that do not have local disk space can ship their log records to the server. In addition, restart recovery for a crashed client may be performed by the server or any other client that has access to the log of this client.

Detailed Description Text (26):
When a node 10, 20, 30, 40 of the exemplary system shown in FIG. 2 wishes to read a page owned by another node and not present in its cache, it sends a request for the page to the owner node. If no other node holds an exclusive lock on the page, the owner node grants the lock and sends a copy of the page to the requester. If some other node holds an exclusive lock on the page, the owner node sends a callback message to that node and waits until that node (a) downgrades/releases its lock and (b) sends the copy of the page present in its buffer pool, if any. Then, the owner node grants the lock and sends the page to the requester.

Detailed Description Text (27):
When a node wants to update a page on which it does not hold an exclusive lock, the node requests an exclusive lock from the owner node. The owner node grants the lock immediately when the page is not locked by any other node. If the page is locked by some nodes, then the owner node sends a callback message to these nodes. Once the owner node receives the acknowledgments to all callback requests, it grants the exclusive lock and sends a copy of the page to the requester, if the requester does not have the page cached in its cache.

Detailed Description Text (28):
Nodes periodically take checkpoints. Each check-point record contains the dirty page table (DPT) and information about the transactions that were active at the time of the checkpointing. The DPT contains entries which correspond to pages that have been modified by local transactions and the updates are not present in the disk version of the database. An entry in the DPT of a node N for a page P contains at least the following fields.

Detailed Description Text (34):
An entry corresponding to a page owned by N is removed from N's DPT when the page is forced to disk. An entry corresponding to a page owned by a remote node is dropped from N's DPT when N receives an acknowledgment from the owner node that the page has been flushed to disk, and the page has not been updated again after the last time it was replaced from the local cache. Dropping an entry for an updated page that is present in the local cache could result in incorrect recovery if N were to crash after taking a checkpoint. This is because the DPT stored in the checkpoint record would not contain an entry for this page.

Detailed Description Text (35):
Transaction rollback is handled by each node. Furthermore, nodes can support the save point concept and offer partial rollbacks. Both total and partial transaction rollbacks open a log scan starting from the last log record written by the transaction. Since updated pages are allowed to be replaced from the node's cache, the rollback procedure may have to fetch some of the affected pages from the owner nodes.

Detailed Description Text (36):
2. Single Node Crash Recover

Detailed Description Text (37):
When a node fails, its lock table and cache contents are lost. As a consequence, any further lock and data requests with respect to the data owned by the failed node are stopped until the node recovers. However, transaction processing on the remaining nodes can continue in parallel with the recovery of the crashed node.

Detailed Description Text (38):
The recovery of a crashed node involves the recovery of updates performed by locally executed transactions. In addition, the recovery of a crashed node may involve the recovery of updates performed by transactions that were executed in another node, referred to as remote transactions. This is because updated pages that are replaced from a node's cache are sent to the owner node. If the failed node does not own any data, the recovery of remote transactions is not required. (For instance, in a client-server environment, the crash of a client does not involve the recovery of transactions that were executed in another client or the server). During its recovery, the crashed node has to (a) determine the pages that may require recovery, (b) identify the nodes involved in the recovery, (c) reconstruct lock information, and (d) coordinate the recovery among the involved nodes.

Detailed Description Text (39):
The following sections (sections a-d) provide details on how the present invention solves the above problems. While providing these details, it is assumed that recovery is carried out by the crashed node when this node restarts. Nevertheless, the present invention allow any node that has access to the database and the log file of the crashed node to perform crash recovery. This is realized in shared disks architectures where all nodes have access to the same database and all log files, as well as in shared nothing and client-server architectures that use hot standby nodes.

Detailed Description Text (41):
When a node fails, all dirty pages present in the cache of this node have to be recovered. These pages belong to two categories: pages owned by the crashed node and pages owned by a remote node. While pages belonging to the first category may have been updated by both local and remote transactions, pages in the second category have been updated only by local transactions.

Detailed Description Text (42):
Since each node writes log records for updates to pages in its own log file, the pages that were updated by local transactions can be determined by scanning the local log starting from

the last complete checkpoint. These pages correspond to the entries in the DPT that is constructed during the analysis phase of the ARIES algorithm. Among these pages, the candidates for recovery are: (a) pages owned by the crashed node that are not present in the cache of any other node, and (b) pages owned by a remote node that were exclusively locked by the crashed node at the time of the crash.

Detailed Description Text (43):
The basic ARIES algorithm cannot be used to determine all dirty pages that belong to the first category. This is because under ARIES, a page is not considered dirty if it is not included in the DPT logged in the last checkpoint before the crash, and no log records for this page are logged after the checkpoint. There are two reasons that a page owned by a node is not considered dirty when it is present in the node's cache at the time of the crash. The first is that the page was updated only by local transactions and it was forced to disk before the checkpoint was taken. This case does not cause any problems because the page is no longer dirty at this point. The second reason is that the page was updated only by remote transactions after the checkpoint was taken and the page was not included in the logged DPT. In this case, no log records for updates to the page are found in the local log file.

Detailed Description Text (44):
However, according to the way each DPT is updated for pages owned by remote nodes, pages that were updated before the crash will have an entry in at least one DPT of the remaining nodes. Among these pages, the pages that may have to be recovered are only those that are present in the DPT of a node and not present in the cache of any other node. The rest of the pages, which are present in the cache of some node, contain all the updates performed on them before the owner node's crash and they do not require recovery. Thus, when the crashed node N restarts, it requests from each operational node N, the list of all pages owned by N that are present in N.sub.r 's cache, as well as all entries in N.sub.r 's DPT that correspond to pages owned by N. After all operational nodes send the above lists to N, N is able to determine the pages that may have to be recovered based on these lists and its own DPT.

Detailed Description Text (45):
But, pages owned by the crashed node that are present in the DPTs of some nodes and the caches of some other nodes may not be recovered at all or recovered incorrectly if a node were to crash after the owner node finishes its restart recovery. Pages that are not in the DPT of the crashed node would not be recovered at all, while pages that are in the DPT would be recovered incorrectly if the disk version of them did not contain all the updates performed by the rest of the nodes in the past. The present invention solves this problem as follows. After the owner node constructs the list of the pages that may have to be recovered, it requests the pages that are present in the cache of a node and have entries in the DPTs of some other nodes from the nodes that have them in their caches. If there are multiple nodes that have the same page in their caches, only one node is notified to send the page.

Detailed Description Text (46):
b. Identifying the Nodes Involved in the Recovery

Detailed Description Text (47):
The crashed node identifies the nodes that are involved in the recovery of a page during the procedure of identifying the pages that require recovery. These nodes belong to two categories: nodes whose DPT entry for P has a PSN value greater than or equal to P's PSN value on disk, and nodes whose DPT entry for P has a PSN value less than P's PSN value on disk. Nodes in the first group have to recover their committed updates. However, some of the nodes in the second group may not have to recover P at all if their log files do not contain any log record that was written for P and whose PSN value is greater than or equal to P's PSN value. This happens when all the updates these nodes made on P took place before P was forced to disk. Thus, a node whose CurrPSN value in its DPT entry for P is less than or equal to P's PSN value is not involved in the recovery process and it can drop P's entry from its DPT.

Detailed Description Text (49):
Before the crashed node N starts recovering the pages that were identified to require recovery, N has to reconstruct its lock information so that normal transaction processing can continue in parallel with the recovery procedure. The lock information includes all the locks that had been granted to both local and remote transactions. The locks that were granted to remote transactions are present in the lock tables of the nodes where those transactions were

executed. In addition, locks that were granted to local transactions for pages owned by remote nodes are also present in the lock tables of the remote nodes.

Detailed Description Text (50):
During restart recovery, each operational node N.sub.r releases all shared locks held by the crashed node. Exclusive locks are retained so that operational nodes are prevented from accessing a page that has not yet been recovered. The list of locks N.sub.r had acquired from the crashed node as well as the list of exclusive locks held by the crashed node are sent to the crashed node. After all the lock lists have been sent, the crashed node can establish its lock tables. In addition, the crashed node needs to acquire exclusive locks for the pages present in its DPT that do not have a lock entry. At this point, all lock tables contain all the needed locks and normal transaction processing can continue.

Detailed Description Text (51):
d. Coordinating the Recovery Among the Involved Nodes

Detailed Description Text (52):
After the crashed node N identifies both the pages that require recovery and the nodes that will participate in the recovery of these pages, the recovery of each page P has to be done in the correct order. This order corresponds to the order in which transactions that were executed at the involved nodes updated P. Since the granularity of locking is a page, only one node can update P at a time. Hence, the PSN values stored in the log records written for P determine the order of updates. In fact, the PSN value stored in the first log record written for P by each transaction that updated P is enough for determining the order of updates. The construction of the above list of PSN values for each page that requires recovery, referred to as NodePSNList, is explained below.

Detailed Description Text (53):
When a remote node N.sub.r receives the list of pages that require recovery from N, it scans its log file starting from the minimum of all RedoLSN values belonging to DPT entries for the pages that are included in the above list. The PSN value present in a log record examined during the scan is inserted into the NodePSNList when (a) the log record corresponds to an update performed on a page present in the above list, and (b) the transaction that wrote the log record is not the same as the transaction that wrote the log record whose PSN field is the last PSN inserted into the NodePSNList, if any. In addition, the location of this log record is remembered and it will be used during the recovery of the page. When the scan is over, N.sub.r sends the NodePSNList to N.

Detailed Description Text (54):
In parallel to the above process, N constructs its own NodePSNList for the pages that require recovery, the same way as the one described above..sup.1 Once all nodes involved in the recovery have sent their NodePSNLists to N, N coordinates the recovery of a page P in the following way.

Detailed Description Text (55):
1. Order the nodes involved in the recovery of P in an ascending ordering based on P's PSN values present in the NodePSNLists sent, including your own NodePSNList. Adjacent entries that correspond to the same node are merged into one entry, whose PSN value is the minimum of the two PSNs.

Detailed Description Text (56):
2. Send P to the node N.sub.r having the minimum PSN entry in the above list. The second minimum PSN value present in the list is also sent to N.sub.r, if any.

Detailed Description Text (59):
When a node receives P from N together with a PSN value, it recovers P by scanning its log starting from either the log record with LSN equal to the RedoLSN value present in the DPT entry for P or the log record remembered in the analysis process mentioned above. The node stops the recovery process when it finds a log record written for P whose PSN value is greater than the PSN value N sent along with P, or when the entire log is scanned. In the former case, the node sends P back to N and remembers the current location in the local log. This location will be the starting point for the continuation of the recovery process for P. In the latter case, the node sends only P back to N. If no PSN value was sent along with P, the node stops

the recovery process when the entire log is scanned.

Detailed Description Text (60):
During the recovery process, nodes update the DPT entries corresponding to pages that are being recovered. In particular, a node that does not apply any log record to a page drops the entry from its DPT when it does not hold a lock on the page. If the node holds a lock on the page, it sets the RedoLSN value of the DPT entry to the current end of the log. The former case is realized when the owner node crashes before acknowledging the writing of the page to disk. The latter case corresponds to the case where all the updates the node performed in the past are present on the disk version of the page and the node has not updated the page since.

Detailed Description Text (61):
3. Multiple Node Crash Recover

Detailed Description Text (62):
So far, details of how the present invention recovers from the case of a single node crash have been described. However, a second node may crash while another node is in the process of recovering from its earlier failure. Recovery from multiple node crashes is similar to the recovery from a single node crash, although it is more expensive as more log files have to be examined and processed and the recovery of a crashed node may have to be restarted. Similar to the single node crash, operational nodes may continue accessing the pages they have in their local caches while the rest of the nodes are in the process of recovering.

Detailed Description Text (63):
As in the single node crash case, the present invention: (a) determines the pages that may require recovery, (b) identifies the nodes that are involved in the recovery, (c) reconstructs the lock information of each crashed node, and (d) coordinates the recovery of a page among the involved nodes. Once the present invention determines the pages that may require recovery, the nodes that are involved in their recovery, the reconstruction of the lock information, and the coordination of the recovery among the involved nodes is done in the same way as in the single node case. Hence, the rest of this section discusses only the solution to the first problem.

Detailed Description Text (64):
As in the single node case, each crashed node has to recover the pages that had been updated by local transactions, as well as the pages that it owns and which had been updated by remote transactions and were present in its cache at the time of the crash. Pages belonging to the first category can be identified from the log records written in the local log file. Unlike the single node crash case, not all pages belonging to the second category can be identified by using only the entries in the DPTs and caches of the operational nodes. The DPTs of the crashed nodes are also needed, for some of these pages may have been updated by several of these nodes.

Detailed Description Text (65):
Although each crashed node lost its DPT during the crash, a superset of each node's DPT can be reconstructed by scanning the node's log file. In particular, each crashed node scans its log by starting from the last complete checkpoint and updates the DPT stored in that checkpoint by inserting new entries for the pages that do not have an entry and are referenced by the examined log records. Once the analysis pass is done, the DPT entries that correspond to pages owned by another node are sent to the owner node. Each operational node also sends the DPT entries that correspond to pages owned by another node and the list of these pages that are present in the local cache to the owner node. The owner node merges all the received entries with the entries it has in its own DPT for the same pages, after removing all entries that correspond to pages cached an operational node. The resulting list corresponds to the pages this node has to recover. Similar to the single node crash, pages present in the cache of an operational node and the DPT of another node are sent to the owner node.

Detailed Description Text (67):
In describing recovery for fine-granularity locking systems, it is assumed that the database is owned by only one node referred to a "server." The rest of nodes are referred to as "clients" and include local disk space. This assumption is made to simplify the description of the present invention by avoiding the need to distinguish between the server and client role of the nodes.

Detailed Description Text (93):
During restart recovery, the crashed client installs in its lock tables the exclusive locks it held before the failure. The recovery of the crashed client involves the recovery of the updates performed by local transactions. Since each client writes all log records for updates to pages in its own log file, all the pages that had been updated before the crash can be determined by scanning the local log starting from the last complete checkpoint. These pages correspond to the entries of the DPT which is constructed during the analysis phase of the ARIES algorithm. However, according to Property 1 and the way the DCT is updated, only the pages that have an entry in the DCT need to be recovered.

Detailed Description Text (95):
During the redo pass, callback log records may be encountered. These callback log records are not processed, according to the discussion presented in Section the beginning of Section V.C. After the redo pass is over, all transactions that were active at the time of the crash are rolled back by using transaction information that was collected during the ARIES analysis pass. Transaction rollback is done by executing the ARIES undo pass.

Detailed Description Text (110):
1. Each client C.sub.i that has P in its cache scans its log and constructs a list, referred to as CallBack.sub.P, of all the objects residing on P that were called back from C. The scan starts from the location corresponding to the RedoLSN value present in the DPT entry about P. CallBack.sub.P contains the object identifiers and the PSN values present in the callback log records written for these objects and the client C. If multiple callback log records are written for the same object and the same client, the PSN value stored in the most recent one is stored in CallBack.sub.P.

Detailed Description Text (112):
Client C installs on P the PSN value sent by the server and starts its recovery procedure for P by examining all log records written for updates to P. The starting point of the log scan is determined from the RedoLSN value present in the DPT entry for P. For each scanned log record, C does the following.

Detailed Description Text (118):
So far, how the present invention recovers for the case of a single client or server crash has been described. However, the server may crash while a client is in the process of recovering from its earlier failure. Similarly, a client may crash while the server is in the process of recovering from its earlier failure. In this case, operational clients will recover their updates on the pages that were present in the server's buffer pool during the crash in the same way as in the server-only crash case. Crashed clients will recover their updates in a way similar to the client-only crash case. In particular, each crashed client will scan its local log starting from the last complete checkpoint and build an augmented DPT. The server will scan its log file, starting from the minimum RedoLSN value present in the DCT stored in the last checkpoint record, and build the DCT entries that correspond to both the pages the crashed clients updated and the pages the operational clients had replaced. From the replacement log records and the PSN value present on each of these pages, the server will calculate the PSN value to be used while recovering those pages in the way explained in Section V.C.3.

Detailed Description Text (120):
Log space management becomes an issue when a node consumes its available log space and it has to overwrite existing log records. Since the earliest log record needed for recovering from a node crash corresponds to the minimum of all the RedoLSN values present in the DPT of this node, the node can reuse its log space only when the minimum RedoLSN is pushed forward. As the present invention has been described so far, the minimum RedoLSN may be pushed forward only when an entry is dropped from the DPT. But, this may not be enough to prevent the node from not having enough log space to continue executing transactions.

Detailed Description Text (121):
The present invention solves the above problem by executing the following steps. When a node replaces a dirty page P from its cache, it remembers the current end of its log. When the owner node forces P to disk, it informs all nodes that had replaced P. These nodes replace the RedoLSN field of the DPT entry referring to P with the remembered end of the log LSN for this page. When a node faces log space problems, it replaces from its cache the page having the minimum RedoLSN value in the DPT and asks the node owning this page to force the page to disk.

If, however, the page is not present in the node's cache, the node just asks the owner node to force the page to disk. If the node needs more log space, it repeats the above procedure. Note that the owner node may be the same as the node that needs to make space in its local log file. In this case, if the page is present in the node's cache, the page is forced to disk. Otherwise, the page is first requested from a node that has it in its cache and then it is written to disk.

Detailed Description Text (122):
As described, the present invention is a distributed database management computer system in which: (1) updated pages are not forced to disk at transaction commit time or when they are replaced from a node cache, (2) transaction rollback and node crash recovery are handled exclusively by the nodes, (3) local log files are never merged during the recovery process, (4) each node can take a checkpoint without synchronizing with the rest of the operational nodes, and (5) clocks do not have to be synchronized across the nodes and lock tables are not checkpointed.

CLAIMS:

1. A method of operating a first node in a distributed database management computer system having a first set of database pages managed by the first node and a second set of database pages managed by a second node, wherein said first node can update pages in the first and second set, comprising the steps of:

(a) updating a database page at the first node;

(b) generating a first log record at the first node;

(c) determining whether the first database page is managed by the first node;

(d) if at step (c) it is determined that the first database page is managed by the first node, writing the first log record to a log storage local to the first node; and

(e) if at step (c) it is determined that the first database page is not managed by the first node:

(e-1) determining whether the first node includes a local log storage;

(e-2) if at step (e-1) it is determined that the first node includes a local log storage, writing the first log record to the local log storage; and

(e-3) if at step (e-1) it is determined that the first node does not include a local log storage, sending the first log record to a node that manages the first database page.

2. The method of claim 1, wherein each of the plurality of nodes includes lock information, further comprising the steps of:

(f) recovering from a failure of the first node by performing the following steps:

(f-1) determining which of the plurality of database pages require recovery;

(f-2) identifying which of the plurality of nodes are involved in the recovery;

(f-3) reconstructing the lock information; and

(f-4) coordinating the recovery among the involved nodes.

3. The method of claim 1, wherein the first node comprises a cache memory for buffering the log record and wherein either step (d) or step (e) is performed when said cache memory is full.

4. The method of claim 3, wherein either step (d) or step (e) is performed when the first database page is written to storage local to the first node.

5. The method of claim 3, wherein step (e) is performed when the first database page is sent to

a second node that manages the first database page.

7. A first node in a distributed database management computer system having a first set of database pages managed by the first node and a second set of database pages managed by a second node, wherein said first node can update pages in the first and second set, comprising:

means for updating a first database page at the first node;

means for generating a first log record at the first node;

first means for determining whether the first database page is managed by the first node;

first means for writing the first log record to a log storage local to the first node if said first means for determining determines that the first database page is managed by the first node;

second means for determining whether the first node includes a local log storage if said first means for determining determines that the first database page is not managed by the first node;

second means for writing the first log record to the local log storage if said second means for determining determines that the first node includes a local log storage; and

means for sending the first log record to a a node that manages the first database page if said second means for determining determines that the first node does not include a local log storage.

8. The computer system of claim 7, wherein each of the plurality of nodes includes lock information, further comprising:

means for recovering from a failure of the first node, said means for recovering comprising:

means for determining which of the plurality of database pages require recovery;

means for identifying which of the plurality of nodes are involved in the recovery;

means for reconstructing the lock information; and

means for coordinating the recovery among the involved nodes.

Previous Doc Next Doc Go to Doc#

Freeform Search

Database: US Pre-Grant Publication Full-Text Database
US Patents Full-Text Database
US OCR Full-Text Database
EPO Abstracts Database
JPO Abstracts Database
Derwent World Patents Index
IBM Technical Disclosure Bulletins

Term: L17 and (delet$ or remov$)

Display: 50 **Documents in Display Format:** - **Starting with Number** 1

Generate: ○ **Hit List** ◉ **Hit Count** ○ **Side by Side** ○ **Image**

Search Clear Interrupt

Search History

DATE: Tuesday, March 01, 2005 Printable Copy Create Case

Set Name side by side	Query	Hit Count	Set Name result set
DB=PGPB,USPT,USOC,EPAB,JPAB,DWPI,TDBD; PLUR=YES; OP=OR			
L18	L17 and (delet$ or remov$)	12	L18
L17	L16 and (persistent)	12	L17
L16	L15 and (dirty)	12	L16
L15	L14 and (log near recovery)	31	L15
L14	(begin$ near recovery) and failure	642	L14
L13	L12 and (recovery near log)	4	L13
L12	(begin$ near recover$) and ((single or multiple) near failure)	23	L12
L11	L10 and ((single or multiple) near failure)	3	L11
L10	l1 and (begin$ near recover$)	65	L10
L9	L8 and (dirty near data)	4	L9
L8	L7 and ((single or multiple) near failure)	40	L8
L7	L6 and log$	1052	L7
L6	L5 and node$	1204	L6
L5	l1 and recovery	2792	L5
L4	L3 and (multiple near failure)	4	L4
L3	L2 and (single near failure)	16	L3
L2	L1 and (data near recovery)	503	L2
L1	707/$.ccls.	25171	L1

Your search matched **72** of **1131693** documents.
A maximum of **500** results are displayed, **15** to a page, sorted by **Relevance** in **Descending** order.

Refine This Search:
You may refine your search by editing the current search expression or entering a new one in the text box.

recovering <and> large <and> distributed Search

☐ Check to search within this result set

Results Key:
JNL = Journal or Magazine **CNF** = Conference **STD** = Standard

---

1 **A recoverable distributed shared memory integrating coherence and recoverability**
Kermarrec, A.-M.; Cabillic, G.; Gefflaut, A.; Morin, C.; Puaut, I.;
Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on , 27-30 June 1995
Pages:289 - 298

[Abstract] [PDF Full-Text (868 KB)] **IEEE CNF**

---

2 **Recovering in large distributed systems with replicated data**
Triantafillou, P.;
Parallel and Distributed Information Systems, 1993., Proceedings of the Second International Conference on , 20-22 Jan. 1993
Pages:39 - 47

[Abstract] [PDF Full-Text (716 KB)] **IEEE CNF**

---

3 **Independent recovery in large-scale distributed systems**
Triantafiliou, P.;
Software Engineering, IEEE Transactions on , Volume: 22 , Issue: 11 , Nov. 1996
Pages:812 - 826

[Abstract] [PDF Full-Text (2220 KB)] **IEEE JNL**

---

4 **Control of robots with elastic joints: deterministic observer and Kalman filter approach**
Timchenko, A.; Kircanski, N.;
Robotics and Automation, 1992. Proceedings., 1992 IEEE International Conference on , 12-14 May 1992
Pages:722 - 727 vol.1

[Abstract] [PDF Full-Text (348 KB)] **IEEE CNF**

---

5 **Microwave image reconstruction utilizing log-magnitude and unwrapped phase to improve high-contrast object recovery**
Meaney, P.M.; Paulsen, K.D.; Pogue, B.W.; Miga, M.I.;
Medical Imaging, IEEE Transactions on , Volume: 20 , Issue: 2 , Feb 2001
Pages:104 - 116

[Abstract] [PDF Full-Text (800 KB)] IEEE JNL

---

6 **Charge trapping and annealing in high-/spl kappa/ gate dielectrics**
Felix, J.A.; Shaneyfelt, M.R.; Fleetwood, D.M.; Schwank, J.R.; Dodd, P.E.; Gusev, E.P.; Fleming, R.M.; D'Emic, C.;
Nuclear Science, IEEE Transactions on , Volume: 51 , Issue: 6 , Dec. 2004
Pages:3143 - 3149

[Abstract] [PDF Full-Text (288 KB)] IEEE JNL

---

7 **Nonuniform sampling techniques for antenna applications**
Rahmat-Samii, Y.; Cheung, R.;
Antennas and Propagation, IEEE Transactions on [legacy, pre - 1988] , Volume: 35 , Issue: 3 , Mar 1987
Pages:268 - 279

[Abstract] [PDF Full-Text (1200 KB)] IEEE JNL

---

8 **Expressnet: A High-Performance Integrated-Services Local Area Network**
Tobagi, F.; Borgonovo, F.; Fratta, L.;
Selected Areas in Communications, IEEE Journal on , Volume: 1 , Issue: 5 , Nov 1983
Pages:898 - 913

[Abstract] [PDF Full-Text (1632 KB)] IEEE JNL

---

9 **Dynamical modeling of cellular response to short-duration, high-intensity electric fields**
Joshi, R.P.; Hu, Q.; Schoenbach, K.H.;
Dielectrics and Electrical Insulation, IEEE Transactions on [see also Electrical Insulation, IEEE Transactions on] , Volume: 10 , Issue: 5 , Oct. 2003
Pages:778 - 787

[Abstract] [PDF Full-Text (1807 KB)] IEEE JNL

---

10 **Optimal time-activity basis selection for exponential spectral analysis: application to the solution of large dynamic emission tomographic reconstruction problems**
Maltz, J.S.;
Nuclear Science, IEEE Transactions on , Volume: 48 , Issue: 4 , Aug. 2001
Pages:1452 - 1464

[Abstract] [PDF Full-Text (344 KB)] IEEE JNL

---

11 **Fluid analysis of arrival routing**
Veatch, M.H.;
Automatic Control, IEEE Transactions on , Volume: 46 , Issue: 8 , Aug. 2001
Pages:1254 - 1257

[Abstract] [PDF Full-Text (132 KB)] IEEE JNL

12 **Correction for the random coincidences in dual-head gamma camera imaging**
Brasse, D.; Comtat, C.; Trebossen, R.; Tararine, M.; Nguyen, Q.T.; Bendriem, B.;
Nuclear Science, IEEE Transactions on , Volume: 48 , Issue: 3 , June 2001
Pages:864 - 871

[Abstract] [PDF Full-Text (144 KB)] IEEE JNL

13 **True single-phase adiabatic circuitry**
Suhwan Kim; Papaefthymiou, M.C.;
Very Large Scale Integration (VLSI) Systems, IEEE Transactions on , Volume: 9 , Issue: 1 , Feb. 2001
Pages:52 - 63

[Abstract] [PDF Full-Text (596 KB)] IEEE JNL

14 **Synchronized chaotic mode hopping in DBR lasers with delayed opto-electric feedback**
Liu, Y.; Davis, P.; Aida, T.;
Quantum Electronics, IEEE Journal of , Volume: 37 , Issue: 3 , March 2001
Pages:337 - 352

[Abstract] [PDF Full-Text (388 KB)] IEEE JNL

15 **Diesel engine exhaust treatment with a pulsed streamer corona reactor equipped with reticulated vitreous carbon electrodes**
Locke, B.R.; Ichihashi, A.; Hyun Ha Kim; Mizuno, A.;
Industry Applications, IEEE Transactions on , Volume: 37 , Issue: 3 , May-June 2001
Pages:715 - 723

[Abstract] [PDF Full-Text (180 KB)] IEEE JNL

1 2 3 4 5 Next

Home | Log-out | Journals | Conference Proceedings | Standards | Search by Author | Basic Search | Advanced Search | Join IEEE | Web Account | New this week | OPAC Linking Information | Your Feedback | Technical Support | Email Alerting | No Robots Please | Release Notes | IEEE Online Publications | Help | FAQ| Terms | Back to Top

L14: Entry 8 of 16 File: USPT Nov 21, 2000

DOCUMENT-IDENTIFIER: US 6151607 A

**** See image for Certificate of Correction ****

TITLE: Database computer system with application recovery and dependency handling write cache

Brief Summary Text (30):

Another approach is to reduce the application state to some manageable size and use a recoverable resource manager to store it. The resource manager might be a database or a recoverable queue. Reducing state size can be facilitated by the use of a scripting language for the application. In this case, the script language interpreter stores the entire application state at well-chosen times so that failures at inappropriate moments survive, and the application execution can continue from the saved point.

Brief Summary Text (31):

Another technique is to use a persistent programming language that logs updates to a persistent state. The idea is to support recoverable storage for processes. When the entire state of the application is contained in recoverable storage, the application itself can be recovered. Recoverable storage has been handled by supporting a virtual memory abstraction with updates to memory locations logged during program execution. If the entire application state is made recoverable, a very substantial amount of logging activity arises. This technique is described in the following publications: Chang and Mergen, "801 Storage: Architecture and Programming," ACM Trans. on Computer Systems, 6, 1 (February 1988) pages 28-50; and Haskin et al., "Recovery Management in QuickSilver," ACM Trans. on Computer Systems, 6,1 (February 1988) pages 82-108.

Brief Summary Text (39):

In the event of a system failure, the database computer system begins with the stable database state and replays the stable log to redo certain logged application operations. The database computer system redoes a logged application operation if its state ID is later in series than the state ID of the most recently flushed or already partially recovered application state.

Brief Summary Text (41):

The read optimizing technique eliminates posting the read values to the log by substituting, for the read values, an identity of the location from where the values are read and posting the identity instead of the values. However, the data is now only available from the read object itself and hence, attention must be paid to the order in which objects are flushed to stable storage. If objects are flushed out of proper sequence, a particular state of an object may be irretrievably lost.

Brief Summary Text (42):

A cache manager has an object table which tracks the objects maintained in the volatile cache. The object table includes fields to track dependencies among the objects. In one implementation, the object table includes, for each object entry, a predecessor field which lists all objects that must be flushed prior to the subject object, and a successor field which lists all objects before which the subject object must be flushed. In another implementation, the object table contains, for each object entry, a node field to store dependencies in terms of their nodes in a write graph formulation.

Brief Summary Text (46):

Still another aspect of this invention is to optimize the recovery procedures invoked following a system crash. During normal operation, each log record is assigned a log sequence number (LSN). The cache manager maintains a recovery log sequence number (rLSN) that identifies the first log record for an associated object at which to begin replaying the operations during recovery. The cache manager occasionally flushes an object to non-volatile memory to install the operations performed on the object. On some occasions, the flushing of one object installs

operations that wrote another data object that has not yet been flushed (i.e., an object that is unexposed in the write graph, meaning that its contents are not needed for recovery). The cache manager advances the rLSN for both objects to identify subsequent log records that reflect the objects at states in which the operations that previously wrote the states are installed in the non-volatile memory.

Drawing Description Text (22):
FIG. 21 is a diagrammatic illustration showing a write graph with a combined node formed from two collapsed nodes, and how the cache manager tracks this event.

Detailed Description Text (45):
An execute operation 92 reads the application state A.sub.1, performs some internal executions, and writes the application state A.sub.2 by means of the application executing beginning in at state A.sub.1 and the execution resulting in state A.sub.2. The resource manager logs the application identifier A, a state ID 2, and the execution operation Ex that resulted in the application state A.sub.2.

Detailed Description Text (55):
Following a system failure, the database computer system invokes a recovery manager 71 to recover the data pages (and other data objects) and application state lost during the crash. During redo recovery, the recovery manager 71 retrieves the most recently flushed data objects and application objects in the stable database and replays the operations in the log against the stable objects. The recovery manager 71 can be implemented as a conventional recovery manager which replays the stable log, beginning at a point known to be earlier than the oldest logged operation that was not yet installed. The recovery manager compares the state ID of each logged operation in the stable log with the state ID of a retrieved data object or application object. If the state ID of the logged operation is later than the state ID of the stable object, the recovery manager redoes that logged operation. This redo process returns the database computer system to the previous state in which it was operating immediately prior to the crash, including the recovered applications.

Detailed Description Text (68):
FIG. 11 shows a cache manager 120 with an object table 122. The object table 122 holds a list of objects that are presently stored, in the volatile cache or that have flush dependencies with objects presently stored. These objects may be in the form of application objects or data objects. Typically, the data objects have volatile (i.e. cache) locations that are identified as memory pages. With regard to data objects, the object table 122 is similar to prior art "page tables." However, unlike prior art page tables, the object table 122 also maintains a list of application objects, with each application object comprising the application address space, and information with each entry that is used to manage flush dependencies.

Detailed Description Text (69):
The object table 122 shows an entry 124 for the application object A and an entry 126 for the data object O which reflect respective object states following the read operation 110. These entries contain information pertaining to the objects which is organized in data structures 128 and 130. Each data structure has an object identifier field 131, 132 to hold the object identifier (e.g., A or O), a state identifier field 133, 134 to hold the state ID for the value of the object, a dirty flag field 135, 136 which holds a flag bit indicating whether or not the object has been modified in volatile cache without those modifications being flushed to stable memory, and a cache location field 137, 138 to hold an address to a location in volatile cache where the current cached value of the object physically resides. The data structure may further have a stable location field to hold an address of the object in stable memory, although this field is not shown in this example. Alternatively, the stable location may be derivable from the object identifier, objectID, in field 131, 132.

Detailed Description Text (71):
FIG. 12 shows a write graph 144 for a read-write edge in which a read operation reads a data object O at a first state during execution of the application object A, and subsequently a write operation writes the data object O to create a second state of that data object. In write graph notation, the circles represent nodes. A write graph node n is characterized by a set of operations ops(n) and a set vars(n) of variables (i.e., objects) written by the operations in ops(n). There is an edge between write graph nodes m and n if there is an installation graph edge between an operation in ops(m) and an operation in ops(n). The cache manager installs the

operations of ops(n) by flushing the objects of vars(n) atomically.

Detailed Description Text (72):
Write graph 144 has two nodes, an application node 146 with vars(146)={A} and a node 148 with vars(148)={O}. The application node 146 shows that the read operation has been performed which changes the application state (by reading values into the application buffers) and that the application has continued its execution with an Ex(A) operation. The data node 148 shows that the write operation affects the object state.

Detailed Description Text (73):
Write graph 144 demonstrates a flush order dependency between the application object and data object. To ensure correct recovery of the application, the cache manager flushes the application object represented by node 146, thereby installing the read operation, prior to flushing the data object represented by node 148.

Detailed Description Text (74):
This write graph further illustrates that, for a logical read operation, an application object A has no predecessor for which it is concerned. All paths between nodes 146 and 148 are at most a length of one. Only the data object O has a predecessor and that predecessor is the application object A (which read it). The logical read operation, by itself, thus reduces to a straightforward result. With reference again to FIG. 11, the predecessor field 140 denotes a list of predecessors for the object O entry 130. The predecessor entry shown contains the identifier for the application object A data record 128, denoted as the predecessor object PO. This predecessor is established when the read operation 110 (FIG. 10) is encountered. The predecessor entry also includes a state identifier for the originating object O, i.e., O.SID. That is, in the general case, an entry on the predecessor list is represented as:

Detailed Description Text (87):
FIGS. 13 shows an alternative construction of the object table. In FIG. 13, the object table 150 contains an entry 152 for a data object O at state 1. This entry includes a data structure 154 having an object identifier field 156, a dirty flag field 158, a cache location field 160, a predecessor field 162, and a successor field 164. In data structure 154, the predecessor field 162 contains an index to a separate predecessor table 166.

Detailed Description Text (91):
In the general recovery scheme introduced at the beginning of this detailed disclosure, a write operation involves posting, to the stable log in association with the write operation, all of the values that are written to an object. The logged operation can be described as simply writing the object state of a data object. This yields a physiological operation that can be handled using conventional cache managers and cache management techniques. The conventional cache manager need not be concerned with object flushing sequence or preserving a certain object state because any data value written to an object, and hence is needed during recovery, is available directly from the stable log.

Detailed Description Text (99):
Corresponding write graphs 202-212 are provided below each operation. The write graphs consist of nodes. Each node n identifies a set of uninstalled operations (i.e., the abbreviations above the dotted line within the nodes), denoted ops(n), in correlation with a set of data or application objects written by the operations (i.e., the abbreviations below the dotted line within the nodes), denoted vars(n). The cache manager usually sees the operations in serialization order. Including the operations in the write graphs in that order is fine because serialization is stronger than installation order.

Detailed Description Text (100):
At the read operation 190, the corresponding write graph 202 consists of a node containing application object A. The read operation 190 reads application state A.sub.1 and data object state O.sub.1 and writes application state A.sub.2. This is reflected in the write graph 202 as involving two nodes: one node containing the application object A and one node containing the data object O. The read operation is registered in the node containing the application object A because the operation writes the application state. The notation R.sub.190 (i.e., read operation 190) in the node containing the application object A indicates that the read operation writes object A. No operation is placed in the node containing object O, because the read operation does not write the object state.

Detailed Description Text (102):
1. Merge into a single node m all nodes n for which vars(n) intersect (write(Op) intersect read (Op)) is not null, where write(Op) is the set of variables written by operation Op, and read (Op) is the set of variables read by Op.

Detailed Description Text (103):
2. If the resulting graph has a cycle, collapse each strongly connected region of the graph into a single node. Each such node n has ops(n) that equals the union of ops(p) of nodes p contained in its strongly connected region and vars(n) that equals the union of vars(p).

Detailed Description Text (104):
3. For each node p.noteq.m, set vars(p)=(vars(p)-nx(Op)). This removes from vars(p) objects that become not exposed, where nx(Op)=write(Op)-Read(Op)

Detailed Description Text (105):
4. To ensure that objects in node p are not exposed when we flush them to install their operations, an edge is defined from each node q with an operation that reads the final version of the object in the node p to node p. Previously, each node p had node q as a potential predecessor.

Detailed Description Text (106):
The read operation 190 introduces a potential read-write edge in write graph 202 from the node containing A to the node containing O. This potential edge (shown as a dashed arrow) indicates that a subsequent write or update of data object O to change its state will create a real edge, thereby establishing a flush order dependency between objects A and O. The direction of the arrow represents the flushing sequence in the flush order dependency. The arrow points from the node containing object A to the node containing object O (i.e., A.fwdarw.O) to represent that the application object A must be flushed before the data object O.

Detailed Description Text (107):
The execute operation 192 reads the application state A.sub.2 and writes the application state A.sub.3. The node containing the object A in the write graph 204 is expanded to include the execute operation (i.e., Ex.sub.192) because the execute operation 196 writes application state A.sub.3. The node containing object O remains void of any operations.

Detailed Description Text (108):
The write operation 194 reads application state A.sub.3 and writes the object state O.sub.2. The write operation is reflected in the write graph 206 by placing the notation W.sub.194 (i.e., write operation 194) in the node containing the data object O. Notice that the write operation 194 does not write the application state, and thus the write operation is not added to the node containing application A.

Detailed Description Text (109):
The write graph 206 also shows a real read-write edge caused by the read and write operations 190 and 194. That is, the previous potential edge has now been converted to a real edge by virtue of the sequence of read-write operations 190 and 194. This read-write edge introduces a flush order dependency between application object A and data object O. To ensure correct recovery of the application, the cache manager must flush the application object A, thereby installing the read operation R.sub.190, prior to flushing the data object O. The read-write edge is indicated by a solid arrow, the direction of which indicates the flushing sequence in the flush order dependency. Here, the application object A must be flushed before the data object O and thus, the arrow points from the node containing object A to the node containing object O (i.e., A.fwdarw.O).

Detailed Description Text (110):
The write operation 194 also introduces a potential edge in write graph 206 from the node containing O to the node containing A. This potential edge indicates that a subsequent write or update of data object A to change its state will create a real edge, thereby establishing a flush order dependency between objects A and O.

Detailed Description Text (111):
The execute operation 196 reads application state A.sub.3 and writes application state A.sub.4.

Since the execute operation 196 writes application object A, the application node A of the write graph 208 is expanded to include that operation (i.e., Ex.sub.196). The execute operation 196 does not write the data object state, and thus the execute operation is not added to the node containing data object O.

Detailed Description Text (112):
The execute operation 196 introduces a real dependency between the data object O and the application object A, as indicated by the write-execute edge. This dependency arises because the data object state O.sub.2 can only be regenerated from values found in the output buffers at application state A.sub.3, which is about to change as a result of the execute operation 196. Since the write optimization technique eliminates logging of the write values to the stable log, the recovery manager must obtain those values from the output buffers of application state A.sub.3 to replay the write operation 194.

Detailed Description Text (113):
To ensure correct recovery of the data object O, the cache manager must flush the data object O, thereby installing the write operation 194 which produces state O.sub.2, prior to flushing the application object A. The write-execute edge is indicated by the solid arrow pointing from the node containing O to the node containing A, thereby indicating an O.fwdarw.A flushing sequence in the flush order dependency.

Detailed Description Text (118):
1. Add each new operation to the intermediate write graph, either including it in a node with existing operations or giving it a node of its own. The intermediate write graph can have cycles.

Detailed Description Text (119):
2. Collapse nodes affected by cycles into a single node n (i.e. all intermediate write graph nodes of the strongly connected region are collapsed into a single write graph node.). The resulting node n has vars(n) consisting of multiple objects.

Detailed Description Text (120):
3. Remove all objects, but one, from the single node. This reduces vars(n) to containing a single object that needs to be flushed in order to install the operations of the node n. The removal of objects can be accomplished through normal write operations, or through a series of blind writes.

Detailed Description Text (121):
These three steps result in a new write graph containing nodes p with vars(p) having a single variable that can be flushed by itself. The edges connecting these nodes impose an order to the flushing of the objects, but the need to atomically flush multiple objects is removed.

Detailed Description Text (125):
2. Intersect the object(s) of step 1 with each set of existing objects associated with a present write graph node n, i.e. objects in vars(n).

Detailed Description Text (126):
3. If all intersections are null, put the operation into its own node.

Detailed Description Text (127):
4. If an intersection is not null, merge all nodes with non-null intersections with the objects of step 1 into a single node.

Detailed Description Text (128):
5. Form edges between intermediate write graph nodes n and m based on when edges exist between the operations of ops(m) and ops(n) in the installation graph.

Detailed Description Text (129):
6. Remove the objects nx(Op)=write(Op)-read(Op) from vars(p) of any other node that currently contains them.

Detailed Description Text (130):
This method is repeated as new operations are executed and the intermediate write graph is

built one operation at a time in operation execution order. A more detailed construction of one exemplary cache manager, and an object table which tracks write dependencies in a manner which effectively handles multi-object nodes and blind write strategies, is described below with reference to FIGS. 18-23.

Detailed Description Text (132):
When a cycle is created, such as the cycle between the nodes containing A and O in the intermediate write graph 208 of FIG. 15, the affected nodes are collapsed into a single node. That is, all intermediate write graph nodes of a strongly connected region are collapsed into a single write graph node. Write graph 210 shows a combined node containing both objects A and O. This combined node contains the union of all operations and objects from the original two nodes. Collapsing intermediate write graph 208 results in the upper node of write graph 210. (The write graph is defined to be acyclic, while the intermediate write graph has cycles.)

Detailed Description Text (133):
Step 3: Reduce Objects In Node to One Object

Detailed Description Text (134):
Forming a combined node containing both A and O has not removed the dependency cycle; rather, both A and O must still be installed atomically together. To break the cycle so that variables can be flushed one by one, all but one object is removed from the node containing multiple objects. This can be done as a result of normal operation, or through a series of blind writes imposed by the cache manager.

Detailed Description Text (135):
With continuing reference to FIG. 15, the read operation 198 involves reading both the data object state O.sub.2 and a new application state B.sub.1, and writing application state B.sub.2. The read operation 198 is reflected in the write graph 210 by addition of a node to contain object B and the inclusion of R.sub.198 (i.e., read operation 198) in that node. Additionally, the read operation 198 introduces a potential read-write edge from the node containing B to the node containing A, O. This potential edge indicates that a subsequent write or update of data object O to change its state will establish a flush order dependency between objects B and O in which the read operation 198 must be installed (by flushing object B) before installation of the operations 190-196.

Detailed Description Text (136):
The write operation 200 reads the application state A.sub.4 and writes object state O.sub.3. The corresponding write graph 212 is expanded to include a third node which contains object O and W.sub.200 (i.e., write operation 200). This operation does not join the existing node containing A,O because write(200) intersect read(200) is s null. The potential read-write edge becomes a real "inverse write-read" edge as a result of this write operation 200. The read operation 198 (R.sub.198) has read the last version of O written by write operation 194 (W.sub.194). This means that a real flush order dependency now exists because data object O's state has been changed in the write operation 200. The flush order dependency dictates that the operation 198 in the node containing object B must be installed prior to the operations 190-196 in the node containing objects A, O. A second flush order dependency is also created by a read-write edge resulting from the write operation. In this dependency, the application object B must be flushed, thereby installing the read operation 198, prior to flushing the data object O.

Detailed Description Text (137):
The purpose of the inverse write-read edge is to ensure that data object O is not exposed when the node with operations 190-196 has no predecessors. This permits the operations 190-196 to be installed by flushing only A.

Detailed Description Text (138):
Notice that the result of write operation 200 removes data object O from the node containing operations 190-196. An object can only reside in one write graph node, which is the last node to write the object. Data object O is in nx(200) and hence is removed from the node containing operations 190-196. Here, the node containing write operation 200 is the last node to write object O, and hence, data object O resides only in that node. No subsequent operation can remove it from that node without also writing it. Because W.sub.194 and W.sub.200 both write data object O, and replay of W.sub.194 does not guarantee the ability to replay W.sub.200,

there is an installation edge from W.sub.194 to W.sub.200. This edge results in a write graph edge from the node with operations 190-196 to the node with operation 200. There is also an edge from R.sub.190 to W.sub.200 so this is a case where a write graph edge results from two installation graph edges.

Detailed Description Text (139):
This is a case in which an object is removed from a multi-object node as a result of normal operation. As a result of the write operation 200, the dependency lo cycle that existed in the intermediate write graph 208 is now broken. That is, a single object A can now be flushed to install all operations 190-196 in the node, including the write operation 194 that originally affected the data object O.

Detailed Description Text (140):
In terms of the write graph, the write operation renders the data object O "unexposed" in the collapsed node of the write graph 212. An "unexposed" object of a write graph node is one that has a write operation for it in a succeeding node and no read operations following the current node that also do not follow the succeeding write. As a result, an unexposed object does not need to be flushed in order to install the operations in the preceding node that wrote that object as no succeeding operation needs the value that it wrote. Conversely, an "exposed" object in a node is an object that needs to be flushed to install the operations in the node that wrote that object. In the FIG. 15 example, the application object A is "exposed" in the collapsed node.

Detailed Description Text (142):
FIG. 16 also shows another technique for reducing the number of objects in a multi-object combined node. The cache manager may not wish to wait for a subsequent write operation of one of the objects in the write graph node, such as write operation 200, because such operations cannot be foreseen and are not guaranteed to occur. Accordingly, the cache manager can impose its own write of an object in the multi-object node. The cache manager performs a "blind identity" write which writes the value of the object onto the stable log. FIG. 16 shows a blind write operation 216 which writes the values of the data object O at state 3, i.e., O.sub.3, to the log record. The blind write creates an after-image of the data object O on the log. That is, the blind write in this case is an identity write because the identical value of data object O, which is the same at both states 2 and 3, is written to the log. The state ID is stepped from 2 to 3 to maintain the convention introduced earlier in this disclosure.

Detailed Description Text (143):
Once the value O.sub.3 is posted to stable log and all nodes that precede the node with operations 190-196 have been installed, i.e. the node with R.sub.198, the cache manager is free to flush the application object A, thereby installing operations 190-196. If the system crashes after object A is flushed and application state A.sub.3 is irretrievably lost, subsequent operations involving the data object O at state 3, can be replayed using the values O.sub.3 on the stable log, rather than the values from the output buffers of a regenerated application state A.sub.3. Blind writes come at a cost of writing larger amounts of data to the log, but this cost is minimal in comparison to the advantages gained by the write optimization techniques in which a high percentage of writes do not result in posting entire object values to the log.

Detailed Description Text (144):
The cache manager-imposed blind write has the same affect of removing an object from the collapsed node in the write graph as a normal write operation. But such a write is under the control of the cache manager, and hence the cache manager can use such writes to help it manage the cache.

Detailed Description Text (145):
FIG. 17 illustrates the effect of a blind write on the combined node in write graph 210 of FIG. 15. In a blind write, the cache manager posts the current value of the data object O to the stable log. This is represented in a write graph 211 as a new node containing the object O and a blind write operation (i.e., W.sub.216). Since the value of O is written to the log, the data object O does not need to be flushed concurrently with the flushing of application object A, and hence the O.fwdarw.A dependency is removed. The blind write thereby breaks the dependency cycle.

Detailed Description Text (146):
In write graph terms, the data object O is no longer "exposed" in the combined node and is withdrawn from that node. The cache manager no longer needs to flush object O as part of the installation of the operations 190-196 in the combined node because it does not matter what object O's value is. The cache manager need only flush the exposed application object A to install all operations in the node, including those that had written data object O, even though data object O is not flushed.

Detailed Description Text (147):
It is noted that, for combined nodes having more than two objects that require simultaneous flushing, the cache manager blind writes all but one object to the stable log.

Detailed Description Text (149):
The object table 222 shows an entry 224 for the data object. The entry is organized in data structure 226 having an object identifier field 228, a dirty flag field 230, a cache location field 232, and a node field 234. The node field contains an index to a separate node list 236 of intermediate write graph nodes. These nodes all write to the object with entry 224. Given that operations write at most one object, an operation can always be associated with exactly one entry in the object table, i.e. the entry whose object it wrote. All intermediate write graph nodes also have operations that write exactly one object. The node list is a list of these intermediate write graph nodes containing operations that write the object table entry.

Detailed Description Text (150):
The node list 236 is a data structure containing various entries 1, . . . , N. Each entry contains a "Last" field 238 that holds data indicating the last update to the object O as a result of the operations of the node. The "Last" field 238 is set to the state identifier of the object at its last update by operations of the node described by node list entry 236. The node list entry also has a node identifier 240 to identify the write graph node into which this intermediate graph node has been collapsed should the node be part of a cycle (a strongly connected region) in the intermediate write graph. In this implementation, the node ID field 240 is an index to a separate node table 246. This data structure contains an entry 248 for write graph nodes that are produced as a result of an intermediate graph collapse. Each such write graph entry has a list of all intermediate graph nodes from which it is constituted via a collapse. These intermediate write graph nodes are identified by pairs <O. O.sid>.

Detailed Description Text (151):
As explained above with reference to FIG. 15, an object can be written by operations in more than one node. The write graph 212 (FIG. 15), for example, shows that data object O, while only requiring flushing in one node, is updated by operations in two different nodes. The node ID fields 240 of all intermediate write graph nodes are "null" until a cycle exists. When a cycle arises, the node ID of the intermediate write graph nodes in the cycle are set to the write graph node identified by entry 248 in the node table 246, which includes the intermediate nodes of the cycle.

Detailed Description Text (152):
Each node list entry in node list 236 further has a predecessor list 242 and a successor list 244. These lists are similar to those described above with respect to FIG. 11 in that they reference predecessor or successor write nodes (in this case, intermediate write graph nodes) which should be flushed before or after the subject node. Each item in the predecessor list 242 or successor list 244 must identify a predecessor or successor node. Since an object can be written by operations in multiple write graph nodes, the object's identifier is no longer sufficient for this node identification. The node can be identified, however, via a pair <object id, state id>, where the state ID is that of the Last attribute for the node at the time the write graph edge was formed. (This can be used in a look up that finds the node with the smallest Last value that is greater than this state ID.) Thus, a node on a predecessor or successor list can be represented by:

Detailed Description Text (154):
The node being referenced in the predecessor and successor lists is an "intermediate node," not the write graph node. Multiple intermediate nodes may comprise a write graph node, which is found from the entries via the Node ID field 240.

Detailed Description Text (155):

A successor list entry need only identify the successor intermediate node by a pair <object id, state id>.

Detailed Description Text (156):
The entries 1-N in the node list 236 are ordered according to the update order sequence. This sequence is readily derived in the data structure by placing the entries in ascending order according to the state identifier in their "Last" field 238.

Detailed Description Text (157):
The cache manager 220 uses the object table 222 to track the flush order dependencies that arise in both read and write operations. Consider the case of a read operation. FIG. 19 shows the read operation 190 from FIG. 15 in more detail. Read operation 190 involves reading both application state A.sub.1 and object state O.sub.1, and writing application state A.sub.2. The intermediate write graph fragment 202 for this operation includes a node 250 containing A and the read operation R.sub.190, and a node 252 containing O without any operations. The read operation 190 results in a potential edge from the node containing A to the node containing O, indicating that a subsequent write or update of data object O to change its state will create a real edge.

Detailed Description Text (158):
As a result of the read operation, the cache manager creates a node entry 254 for the data object O's node list 236 which recognizes object A as a predecessor. Entry 256 is only a "potential" node list entry at this point since a write graph node technically only exists when uninstalled operations write into variables. That is, the node containing data object O becomes a write graph node in write graph 206 following the write operation 194. A node is shown in FIGS. 15 and 19 to help describe how data object O is handled.

Detailed Description Text (159):
More particularly, node list entry 256 has a "Last" field 238 set to "1," the state ID of data object O's last update, and a node ID field set to "null", indicating that this node has not taken part in a "collapse". The predecessor list 242 is updated to reference the predecessor application object A. This node reference is includes the predecessor object ID "A," and A's state ID of 2. In addition, to determine when this edge is real or potential, the node reference includes "firstr(<A,2>,O)," indicating the state ID of data object O when first read by application object A in this node, which is 1. The edge is real only if data object O has a state ID that is greater than 1. Nothing is placed in the successor list 244.

Detailed Description Text (160):
Similarly, the cache manager creates a node entry 256 for the application object A's node list which recognizes data object O as a successor. Entry 256 contains in its "Last" field 238' the state ID of "2" for the application object A's last update and in the node ID field 240' it contains the value null, indicating that this intermediate write graph node is not part of a cycle and hence has not taken part in a collapse. The successor list 244' of entry 256 is updated to reference the successor data object O. This successor reference to identify the node for object O includes the successor object ID "O." and O's state ID of 1. Nothing is placed in the predecessor list 242'.

Detailed Description Text (161):
Next, consider the case of the write operation. FIG. 20 shows the write graph 208 following the execute operation 196 and write operation 194 from FIG. 15 in more detail. Write operation 194 involves reading application state A.sub.3 and writing object state 2. Execute operation 196 involves reading application state A.sub.3 and writing application state A.sub.4. The write graph 208 as a result of this operation includes the node 251 containing A and the node 253 containing O. The write operation 194 represented in node 253 changed the data object state from state O.sub.1 to state O.sub.2, thereby changing the previous "potential" edge to a "real" edge (as represented by the solid arrow). This correlates to object A becoming a real predecessor to object O. Additionally, recall in the write graph 206 of FIG. 15, a second potential edge had been created as a result of the write operation 194 because data object O, to be replayed, must obtain values from application object A at state 3. This successor edge becomes real in write graph 208 of FIG. 20 because the downstream execute operation 196 changes the application state from state 3 to state 4. Thus, the write graph 208 in FIG. 20 shows two real edges between the nodes 251 and 253.

Detailed Description Text (162):
The old entry 254 representing a potential write graph node for data object O is replaced by a real write graph node list entry 262. Entry 262 for data object O is created in response to the writing of data object O at operation 194. The entry 262 has a "Last" field 238" set to the object O's state ID following the write operation 194 (i.e., state ID=2), and a node ID field 240" set to null. The predecessor list 242" in entry 262 includes the same reference to predecessor object A as is contained in the predecessor list 242 in entry 254. The successor list 244" in entry 262 is updated to reference the successor object A. This reference includes the successor object ID "A" and A's state ID of 3. Whether a successor is considered "potential" or "real" has little impact. When the predecessor is flushed, the predecessor is removed from its successors' predecessor list entries, regardless of whether it is real or potential.

Detailed Description Text (163):
With respect to the node list entry 256 for application object A, the "Last" field 238' has been updated to reflect a state 4 since this is the state at the execute operation 196. (FIG. 20 shows the data structures after this operation, but before the collapse of the cycle that is now present.) The cache manager also updates the predecessor list 242' of the node list entry 256 for application object A to reference the "potential" predecessor object O. This node reference includes the predecessor object ID "O," and O's state ID of 2. In addition, to determine when this edge is real or potential, the node reference includes "firstr(<O,2>,A)," indicating the state ID of application object A when first read to write the data object O at state 2, which is a state ID of 3. The edge is real only if application object A has a state ID that is greater than 3. FIG. 20, because it shows the write graph 208 after the execute operation Ex.sub.196, shows the edge as real, with the Last field of 256 set to 4.

Detailed Description Text (164):
Notice that the node list entry 256 for application object A references node list entry 262 of data object O as both a predecessor and a successor. This correlates to cycle dependency in that the data object O must be flushed both before (or not later than) and after (or not earlier than) application object A.

Detailed Description Text (165):
The cache manager recognizes this cyclic condition when it occurs, or when the cache manager goes to flush the application object A. For purposes of continuing discussion, suppose the cache manager decides to flush the application object A. The cache manager proceeds down A's node list, which contains the single entry 256, and discovers the cycle dependency. When a cycle between the intermediate write graph nodes 251 and 253 is discovered, the nodes 251 and 253 are collapsed into a single node.

Detailed Description Text (166):
FIG. 21 shows a write graph 209 having a combined write graph node 255 formed by collapsing nodes 251 and 253 into one node following the execute operation 196 (i.e., EX.sub.196). The node ID field 240" of object O's node list entry 262 is switched from "null" to reference an entry 257 in the node table 246. Additionally, the node ID field 240' of object A's node list entry 256 is changed from "null" to reference the entry 257. The node table entry 257 lists all intermediate graph nodes (identified by pairs <Object, Object State ID>) from which it is constituted via the collapse. In this example, the node table entry 257 identifies the node 251 as <A, 4> and the node 253 as <O, 2>.

Detailed Description Text (167):
To break the cycle dependency and flush the object A by itself, the cache manager first installs all write graph nodes preceding the object A. In this case, the only real predecessor node (which is a node of the intermediate write graph) contains object O, which forms the cycle dependency with A and hence is to be flushed simultaneously with the application object A. The cache manager then blindly writes the data object O listed in the predecessor list 242' of object A's node list entry 256 to the stable log. That is, the values of the data object at state 2 (i.e., O.sub.2) are posted to the stable log. This is shown in FIG. 16 as the blind write 216, which results in a log record containing the value O.sub.2.

Detailed Description Text (168):
FIG. 22 shows the blind write operation 216 of data state O.sub.3 and a corresponding write graph 211. The write graph 211 contains three nodes: a node 259 containing exposed object A and

unexposed object O, a node 261 containing exposed object B, and a node 263 containing exposed object O. As a result of the write operation 216, a second entry 264 is added to the node list for data object O. This second entry 264 has a last field 238"" set to a state ID of 3, a node ID field 240"" set to null, a predecessor list field 242"" set to reference the node 261 containing application object B as a real predecessor node, and a successor list field 244"" set to null. The node list entry 256 for object A is also updated following the write operation 216. The last field 238' has been updated to A's last state ID of 4, and the predecessor field 242' is updated to identify the node 261 containing application object B as a predecessor node.

Detailed Description Text (169):
Notice that the node ID fields in A's node list entry 256 and O's node list entry 262 remaining pointing to entry 257 in the node table 246. The cycles have not yet disappeared. The node for data object O in the cycle is no longer the last node for object O, so object O is not in vars (257). But the operations that previously wrote data object O are still in node 259, and this is what is captured by having the node IDs continue to reference 257. The blind write operation 216 rendered object O "unexposed" in node 259 and creates a new intermediate write graph node 263 for data object O.

Detailed Description Text (170):
A node list entry 266 for the application object B is also shown in FIG. 22. This entry 266 reflects the node 261 that was created by the read operation 198 in FIG. 16, prior to the blind write operation 216. The object B's node list entry 266 has a last field 238'" set to B's last state ID of 2, a node ID field 240'" set to null, a predecessor list field 242'" set to null, and a successor list field 244'" set to identify the nodes 259 and 263.

Detailed Description Text (172):
Suppose the cache manager wishes to flush application object A. Before doing that, the node containing A must not have predecessors in the write graph. Thus, the cache manager must first flush B to remove B's node 266 from the write graph. Next, the cache manager flushes the application object A, thereby installing the operations 190-196 contained in node 259 of FIG. 22, which is represented by node table entry 257 of FIG. 21. FIG. 23 shows the results of flushing application object A. The object O's node list entry 262 which contains reference to the node 259 via node table entry 257 that it references via its node ID field is removed as these states are now installed. The successor list field 244' in A's entry 256 is updated to remove all successors since A has now been installed. That the flushing of A leaves it's node list entry 256 with no successors. Accordingly, this flushing operation removes the intermediate graph cycle dependency as the node list entry 256 for application object A no longer contains reference to data object O in either the successor or predecessor list fields.

Detailed Description Text (176):
During recovery, the database computer system can invoke a conventional recovery manager to recover the application state and object state at the instance of the crash. The conventional recovery manager retrieves the most recently flushed data objects and application objects in the stable database. The recovery manager . then replays the stable log, beginning at a point known to be earlier than the oldest logged operation that was not yet installed. For this conventional physiological operation recovery, the recovery manager compares the state ID of each logged operation in the stable log with the state ID of a retrieved data object or application object. If the state ID of the logged operation is later than the state ID of the stable object, the recovery manager redoes that logged operation.

Detailed Description Text (181):
At the time that data object O is flushed, the cache manager marks object O as clean (the dirty flag is reset) in the cache. When O is updated at log record 274, the cache manager sets a recovery log sequence number (rLSN) to identify the log record 274 as the starting point for replay of object O during recovery.

Detailed Description Text (183):
A shortcoming of this conventional recovery technique is that the recovery manager can end up replaying many operations that are unnecessary for recovery. As an example, the lifetimes of some application objects and data objects tend to be short and once terminated or deleted the objects no longer need recovery. If a system failure occurs after an object has terminated, but while that object's updates remain on the active log tail, the recovery manager still redoes

the operations for that object starting from the last stable version of the object. If the object's state was never written to stable memory, all updates reflected in the log records are redone. Unfortunately, the replayed operations for these terminated or deleted objects are unnecessary, and can add substantially to recovery time.

Detailed Description Text (185):
Recall the discussion from FIGS. 16 and 17. A blind write operation posted the value of data object O to the stable log and thereby rendered the data object O "unexposed" in the write graph node containing application object A, meaning that the prior value of O was no longer needed at the time when the write graph node is installed. The flushing of application object A installed all operations (i.e., R.sub.190, EX.sub.192, W.sub.194, and Ex.sub.196), including the write operation W.sub.194 that had written the data object O, even though the data object O itself was not flushed.

Detailed Description Text (186):
FIG. 25 shows an example of the recovery optimization technique for the same stable log 270. Suppose that log record 278 represents a blind write operation in which the cache manager posts the values of data object O at state ID of "n+h" (i.e., O.sub.n+h) to the stable log. The blind write renders the data object O "unexposed" in the write graph node containing both objects A and O, as described above with respect to FIG. 17.

Detailed Description Text (187):
Sometime after the blind write operation, the cache manager flushes the "exposed" application object A at state "m" (i.e., A.sub.m) to install all operations in the write graph node, including any operations that have written the data object O. The blind write and subsequent flushing of application object A renders all operations that wrote the "unexposed" data object O as part of the operations associated with the node for application object A (e.g., log records 274 and 276) unnecessary for recovery.

Detailed Description Text (188):
The cache manager advances the rLSN.sub.A for the "exposed" application object A (not shown in this figure) because all preceding operations affecting A are now installed, akin to the customary case shown in FIG. 24. Similarly, the cache manager advances the rLSN.sub.O for the "unexposed" data object O from its original point at log record 274 to the new location after log record 278 as if the unexposed data object O had also been flushed. Record 278 is the next log record, after the records for the installed operations, that contains an operation writing data object O. The rLSN of object O is logged as a record 284 to reference the log record 278 with the log sequence number of n+i. In this manner, the recovery manger treats "unexposed" objects as if they had been flushed as of their last update in the write graph node being installed by the flushing of the node's exposed variable(s). By logging the rLSN, recovery for O can begin at log record 278.

Detailed Description Text (191):
During recovery, the recovery manager 71 performs two passes: (1) an analysis pass and (2) a redo pass. During the analysis pass, the recovery manager scans the active log tail to locate the rLSNs for all objects. In this example, the rLSN.sub.O for data object O references an LSN of n+i for log record 278. The recovery manager next identifies the minimum recovery log sequence number rLSN.sub.min, similar to the conventional method described above. However, because the rLSNs have been advanced using the recovery optimization techniques, the rLSN.sub.min could be much later in the log as compared to the conventional recovery method, thereby avoiding the replay of operations that are unnecessary for recovery.

Detailed Description Text (192):
During the redo pass, the recovery manager examines all operations on the log beginning at the rLSN.sub.min. More particularly, the recovery manager performs the following redo test for each log record in the stable log that follows rLSN.sub.min :

Previous Doc Next Doc Go to Doc#

Freeform Search

Database: US Pre-Grant Publication Full-Text Database
US Patents Full-Text Database
US OCR Full-Text Database
EPO Abstracts Database
JPO Abstracts Database
Derwent World Patents Index
IBM Technical Disclosure Bulletins

Term: L17 and (delet$ or remov$)

Display: 50 **Documents in Display Format:** - **Starting with Number** 1

Generate: ○ **Hit List** ◉ **Hit Count** ○ **Side by Side** ○ **Image**

Search Clear Interrupt

Search History

DATE: Tuesday, March 01, 2005 Printable Copy Create Case

Set Name side by side	Query	Hit Count	Set Name result set
DB=PGPB,USPT,USOC,EPAB,JPAB,DWPI,TDBD; PLUR=YES; OP=OR			
L18	L17 and (delet$ or remov$)	12	L18
L17	L16 and (persistent)	12	L17
L16	L15 and (dirty)	12	L16
L15	L14 and (log near recovery)	31	L15
L14	(begin$ near recovery) and failure	642	L14
L13	L12 and (recovery near log)	4	L13
L12	(begin$ near recover$) and ((single or multiple) near failure)	23	L12
L11	L10 and ((single or multiple) near failure)	3	L11
L10	l1 and (begin$ near recover$)	65	L10
L9	L8 and (dirty near data)	4	L9
L8	L7 and ((single or multiple) near failure)	40	L8
L7	L6 and log$	1052	L7
L6	L5 and node$	1204	L6
L5	l1 and recovery	2792	L5
L4	L3 and (multiple near failure)	4	L4
L3	L2 and (single near failure)	16	L3
L2	L1 and (data near recovery)	503	L2
L1	707/$.ccls.	25171	L1

END OF SEARCH HISTORY

Hit List

Clear | Generate Collection | Print | Fwd Refs | Bkwd Refs | Generate OACS

Search Results - Record(s) 1 through 16 of 16 returned.

☐ 1. Document ID: US 20040030703 A1

Using default format because multiple data bases are involved.

L14: Entry 1 of 16 File: PGPB Feb 12, 2004

PGPUB-DOCUMENT-NUMBER: 20040030703
PGPUB-FILING-TYPE: new
DOCUMENT-IDENTIFIER: US 20040030703 A1

TITLE: Method, system, and program for merging log entries from multiple recovery log files

PUBLICATION-DATE: February 12, 2004

INVENTOR-INFORMATION:

NAME	CITY	STATE	COUNTRY	RULE-47
Bourbonnais, Serge	Palo Alto	CA	US	
Hamel, Elizabeth Belva	Morgan Hill	CA	US	
Lindsay, Bruce G.	San Jose	CA	US	
Liu, Chengfei	Rostrevor	CA	AU	
Stankiewitz, Jens	Dorsten		DE	
Truong, Tuong Chanh	San Jose		US	

US-CL-CURRENT: 707/100

Full | Title | Citation | Front | Review | Classification | Date | Reference | Sequences | Attachments | Claims | KWIC | Draw Desc | Image

☐ 2. Document ID: US 20020099729 A1

L14: Entry 2 of 16 File: PGPB Jul 25, 2002

PGPUB-DOCUMENT-NUMBER: 20020099729
PGPUB-FILING-TYPE: new
DOCUMENT-IDENTIFIER: US 20020099729 A1

TITLE: Managing checkpoint queues in a multiple node system

PUBLICATION-DATE: July 25, 2002

INVENTOR-INFORMATION:

NAME	CITY	STATE	COUNTRY	RULE-47
Chandrasekaran, Sashikanth	Bellmont	CA	US	
Bamford, Roger J.	Woodside	CA	US	
Bridge, William H.	Alameda	CA	US	
Brower, David	Alamo	CA	US	
MacNaughton, Neil	Los Gatos	CA	US	
Chan, Wilson Wai Shun	San Mateo	CA	US	

Srihari, Vinay San Francisco CA US

US-CL-CURRENT: 707/203

Full | Title | Citation | Front | Review | Classification | Date | Reference | Sequences | Attachments | Claims | KWIC | Draw Desc | Image

---

☐ 3. Document ID: US 20020091718 A1

L14: Entry 3 of 16 File: PGPB Jul 11, 2002

PGPUB-DOCUMENT-NUMBER: 20020091718
PGPUB-FILING-TYPE: new
DOCUMENT-IDENTIFIER: US 20020091718 A1

TITLE: METHOD AND APPARATUS FOR DETECTING AND RECOVERING FROM DATA CORRUPTION OF DATABASE VIA READ LOGGING

PUBLICATION-DATE: July 11, 2002

INVENTOR-INFORMATION:

NAME	CITY	STATE	COUNTRY	RULE-47
BOHANNON, PHILIP L.	MORRIS	NJ	US	
RASTOGI, RAJEEV	UNION	NJ	US	
SESHADRI, SRINIVASAN	BASKING RIDGE	NJ	US	
SILBERSCHATZ, ABRAHAM	UNION	NJ	US	
SUDARSHAN, SUNDARARAJARAO	POWAI, BOMBAY		IN	

US-CL-CURRENT: 707/202

Full | Title | Citation | Front | Review | Classification | Date | Reference | Sequences | Attachments | Claims | KWIC | Draw Desc | Image

---

☐ 4. Document ID: US 6701345 B1

L14: Entry 4 of 16 File: USPT Mar 2, 2004

US-PAT-NO: 6701345
DOCUMENT-IDENTIFIER: US 6701345 B1

TITLE: Providing a notification when a plurality of users are altering similar data in a health care solution environment

DATE-ISSUED: March 2, 2004

INVENTOR-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY
Carley; Kevin W.	Atlanta	GA		
Harrington; Lisa Marie	Denver	CO		
Dikeman; Jennifer Scot	Atlanta	GA		
Moody; Megan Davies	Denver	CO		
Gregory; Mary Michelle	Atlanta	GA		

ASSIGNEE-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY	TYPE CODE
Accenture LLP	Palo Alto	CA			02

APPL-NO: 09/ 549237 [PALM]
DATE FILED: April 13, 2000

INT-CL: [07] G06 F 15/16

US-CL-ISSUED: 709/205; 709/232, 707/8
US-CL-CURRENT: 709/205; 707/8, 709/232

FIELD-OF-SEARCH: 709/200-202, 709/204, 709/205, 709/203, 709/232, 707/201, 707/203, 707/8

PRIOR-ART-DISCLOSED:

U.S. PATENT DOCUMENTS

PAT-NO	ISSUE-DATE	PATENTEE-NAME	US-CL
3670310	June 1972	Bharwani et al.	
5146600	September 1992	Sugiura	
5220657	June 1993	Bly et al.	711/152
5515491	May 1996	Bates et al.	345/754
5546580	August 1996	Seliger et al.	707/8
5715397	February 1998	Ogawa et al.	
5721943	February 1998	Johnson	
5787450	July 1998	Diedrich et al.	
5805900	September 1998	Fagen et al.	
5859972	January 1999	Subramaniam et al.	
5903889	May 1999	De la Huerga et al.	
5909570	June 1999	Webber	
5933824	August 1999	Dekoning et al.	
5974389	October 1999	Clark et al.	705/3
6088702	July 2000	Plantz et al.	707/103
6192381	February 2001	Stiegemeier et al.	
6240414	May 2001	Beizer et al.	707/8
6438548	August 2002	Grim et al.	707/8

FOREIGN PATENT DOCUMENTS

FOREIGN-PAT-NO	PUBN-DATE	COUNTRY	US-CL
0578406	January 1994	EP	
WO 00 00909	January 2000	WO	

OTHER PUBLICATIONS

Michael Gertz: "Oracle/Sql Tutorial", Database and Information Systems Group, University of California, Davis, Online! Jan. 2000 XP002189251, Retrieved from the Internet: <URL:http://www.db.cs.ucdavis.edu/teaching/sqltutorial/tutorial.pdf> retrieved on Feb. 5, 2002, pag9, 1.4.1 Insertions; pag 58, System architecture.
Per Cederqvist et al.: "Version Management with CVS" Signum Support AB, Online, 1993, pp. I-II, 1-6, 57-66, XP002199439, Retrieved from the Internet: <URL:http://www.loria.fr / {molli/cvs/doc/cvs.ps>retrieved on May 21, 2002, 1.1 What is CVS?, 10 Multiple developers (pag

57-66).

ART-UNIT: 2153

PRIMARY-EXAMINER: Maung; Zarni

ASSISTANT-EXAMINER: Winters; Mareisha N.

ATTY-AGENT-FIRM: Oppenheimer Wolff & Donnelly LLP

ABSTRACT:

A notification when multiple users attempt to alter the same data may first begin when connections to a plurality of user stations are monitored. An instruction for initiating a load process is received from a user station. Data is downloaded from the one of the user stations to the server. It is determined whether another load process is being concurrently executed by another user station. If it is determined that a load process is being concurrently executed, a notification is sent to the user station. A notification is also sent to the user station that initiated the concurrently executing load process. At least one of the load processes is suspended upon detecting the concurrently executed load process. At least one of the load processes may be allowed to continue upon receiving a command to continue from the user station associated with the suspended load process.

18 Claims, 21 Drawing figures

☐ 5. Document ID: US 6490594 B1

L14: Entry 5 of 16 File: USPT Dec 3, 2002

US-PAT-NO: 6490594
DOCUMENT-IDENTIFIER: US 6490594 B1
**** See image for Certificate of Correction ****

TITLE: Database computer system with application recovery and dependency handling write cache

DATE-ISSUED: December 3, 2002

INVENTOR-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY
Lomet; David B.	Redmond	WA		

ASSIGNEE-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY	TYPE CODE
Microsoft Corporation	Redmond	WA			02

APPL-NO: 09/ 566699 [PALM]
DATE FILED: May 8, 2000

PARENT-CASE:
RELATED APPLICATIONS This is a continuation of U.S. patent application Ser. No. 09/301,516, which was filed Apr. 28, 1999, now U.S. Pat. No. 6,151,607, which is a continuation of U.S. patent application Ser. No.08/832,870, filed Apr. 4 1997, which is now U.S. Pat. No. 6,067,550; both filed in the name of David B. Lomet and assigned to Microsoft Corporation.

INT-CL: [07] G06 F 17/00

US-CL-ISSUED: 707/200; 707/201, 707/202
US-CL-CURRENT: 707/200; 707/201, 707/202

FIELD-OF-SEARCH: 707/3-6, 707/200-206, 707/101-103, 707/10, 709/218, 717/108, 717/124

PRIOR-ART-DISCLOSED:

U.S. PATENT DOCUMENTS

PAT-NO	ISSUE-DATE	PATENTEE-NAME	US-CL
4498145	February 1985	Baker et al.	707/202
5257369	October 1993	Skeen et al.	395/650
5287501	February 1994	Lomet	395/600
5317731	May 1994	Dias et al.	
5325528	June 1994	Klein	395/650
5371889	December 1994	Klein	395/650
5412801	May 1995	de Remer et al.	
5485608	January 1996	Lomet et al.	
5524205	June 1996	Lomet et al.	
5594863	January 1997	Stiles	
5701480	December 1997	Raz	395/671
5778388	July 1998	Kawamura et al.	707/203
5806065	September 1998	Lomet	
5819304	October 1998	Nilsen et al.	711/5
5826089	October 1998	Ireton	712/209
5832508	November 1998	Sherman et al.	707/200
5857207	January 1999	Lo et al.	707/203
6044379	March 2000	Callsen	707/10
2002/0062475	May 2002	Iborra et al.	717/108
2002/0091995	July 2002	Arnold et al.	717/124
2002/0095479	July 2002	Schmidt	709/218

OTHER PUBLICATIONS

Cao et al., A delay-optimal quorum-based mutual exclusion algorithm for distributed systems, Parallel and Distributed Systems, IEEE Transactions on, vol. 12, Issue 12, Dec. 2001, pp. 1256-1268.*
Auletta et al., Optimal tree access by elementary and composite templates in parallel memory systems, Parallel and Distributed Systems, IEEE Transactions on, vol. 13, Issue 4, Apr. 2002, pp. 399-412.*
Li et al., Parallel matrix multiplication on a linear array with a reconfigurable pipelined bus system, Computers, IEEE Transactions on, vol. 50, Issue 5, May 2001, pp. 519-525.*
Lomet, David et al., "Redo Recovery after System Crashes", Proceedings of the 21th VLDB Conference, Zurick, Switzerland, 1995, consisting of 14 pages.
Lomet, David et al., "Persistent Applications Using Generalized Redo Recovery," consisting of 10 pages.
C. Mohan et al., "ARIES:A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging" ACM Trans. On Database Systems 17,1 (Mar. 1992) 94-162.

ART-UNIT: 2171

PRIMARY-EXAMINER: Jung; David

ATTY-AGENT-FIRM: Lee & Hayes, PLLC

ABSTRACT:

This invention concerns a database computer system and method for making applications recoverable from system crashes. The application state (i.e., address space) is treated as a single object which can be atomically flushed in a manner akin to flushing individual pages in database recovery techniques. To enable this monolithic treatment of the application, executions performed by the application are mapped to logical loggable operations which can be posted to the stable log. Any modifications to the application state are accumulated and the application state is periodically flushed to stable storage using an atomic procedure. The application recovery integrates with database recovery, and effectively eliminates or at least substantially reduces the need for check pointing applications. In addition, optimization techniques are described to make the read, write, and recovery phases more efficient.

8 Claims, 27 Drawing figures

☐ 6. Document ID: US 6449623 B1

L14: Entry 6 of 16 File: USPT Sep 10, 2002

US-PAT-NO: 6449623
DOCUMENT-IDENTIFIER: US 6449623 B1

TITLE: Method and apparatus for detecting and recovering from data corruption of a database via read logging

DATE-ISSUED: September 10, 2002

INVENTOR-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY
Bohannon; Philip L.	Morris	NJ		
Rastogi; Rajeev	Union	NJ		
Seshadri; Srinivasan	Basking Ridge	NJ		
Silberschatz; Abraham	Union	NJ		
Sudarshan; Sundararajarao	Bombay			IN

ASSIGNEE-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY	TYPE CODE
Lucent Technologies Inc,	Murray Hill	NJ			02

APPL-NO: 09/ 207927 [PALM]
DATE FILED: December 9, 1998

PARENT-CASE:
This application claims priority and is related by subject matter to U.S. application Ser. No. 08/767,048, entitled "System And Method For Restoring A Multiple Checkpointed Database In View Of Loss Of Volatile Memory" of Bohannon et al., filed Dec. 16, 1996, issued Jan. 26, 1999 (now U.S. Pat. No. 5,864,849) and to U.S. provisional patent application Ser. Nos. 60/099,265 and 60/099,271, filed Sep. 4, 1998 of Bohannon et al.

INT-CL: [07] G06 F 12/00

US-CL-ISSUED: 707/202

US-CL-CURRENT: 707/202

FIELD-OF-SEARCH: 707/200, 707/202, 707/204

PRIOR-ART-DISCLOSED:

U.S. PATENT DOCUMENTS

PAT-NO	ISSUE-DATE	PATENTEE-NAME	US-CL
4438494	March 1984	Budde et al.	714/2
5278982	January 1994	Daniels et al.	395/600
5504858	April 1996	Ellis et al.	714/6
5524205	June 1996	Lomet et al.	395/182.14
5581690	December 1996	Ellis et al.	395/182.04
5684944	November 1997	Lubbers et al.	395/182.04
5727206	March 1998	Fish et al.	395/618
5734894	March 1998	Adamson et al.	395/616
5758150	May 1998	Bell et al.	395/610
5778387	July 1998	Wilkerson et al.	707/202
5778395	July 1998	Whiting et al.	707/204
5832526	November 1998	Schuyler	707/205
5845292	December 1998	Bohannon et al.	707/202
5857208	January 1999	Ofek	707/204
5864849	January 1999	Bohannon et al.	707/8
5870758	February 1999	Bamford et al.	707/201
5872966	February 1999	Burg	709/213
5978791	November 1999	Farber et al.	707/2
6016500	January 2000	Waldo et al.	707/202
6041423	March 2000	Tsukerman	714/19
6189011	February 2001	Lim et al.	707/102

ART-UNIT: 2161

PRIMARY-EXAMINER: Trammell; James P.

ASSISTANT-EXAMINER: Wang; Mary

ATTY-AGENT-FIRM: Morgan & Finnegan

ABSTRACT:

A method of detecting and recovering from data corruption of a database is characterized by the step of logging information about reads of a database in memory to detect errors in data of the database, wherein said errors in data of said database arise from one of bad writes of data to the database, of erroneous input of data to the database by users and of logical errors in code of a transaction. The read logging method may be implemented in a plurality of database recovery models including a cache-recovery model, a prior state model a redo-transaction model and a delete transaction model. In the delete transaction model, it is assumed that logical information is not available to allow a redo of transactions after a possible error and the effects of transactions that read corrupted data are deleted from history and any data written by a transaction reading Ararat data is treated as corrupted.

60 Claims, 8 Drawing figures

☐ 7. Document ID: US 6374264 B1

L14: Entry 7 of 16 File: USPT Apr 16, 2002

US-PAT-NO: 6374264
DOCUMENT-IDENTIFIER: US 6374264 B1

TITLE: Method and apparatus for detecting and recovering from data corruption of a database via read prechecking and deferred maintenance of codewords

DATE-ISSUED: April 16, 2002

INVENTOR-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY
Bohannon; Philip L.	Morris	NJ		
Rastogi; Rajeev	Union	NJ		
Seshadri; Srinivasan	Basking Ridge	NJ		
Silberschatz; Abraham	Summit	NJ		
Sudarshan; Sundararajarao	Bombay			IN

ASSIGNEE-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY	TYPE CODE
Lucent Technologies Inc.	Murray Hill	NJ			02

APPL-NO: 09/ 207926 [PALM]
DATE FILED: December 9, 1998

PARENT-CASE:
This application claims priority and is related by subject matter to U.S. application Ser. No. 08/767,048, entitled "System And Method For Restoring A Multiple Checkpointed Database In View Of Loss Of Volatile Memory" of Bohannon et al., filed Dec. 16, 1996 (now U.S. Pat. No. 5,864,849) and to U.S. provisional patent application Ser. Nos. 60/099,265 and 60/099,271, filed Sep. 4, 1998 of Bohannon et al.

INT-CL: [07] G06 F 17/30, G06 F 11/20

US-CL-ISSUED: 707/202; 707/2, 714/13
US-CL-CURRENT: 707/202; 707/2, 714/13

FIELD-OF-SEARCH: 707/1-2, 707/202, 707/7-8, 707/201, 714/13-20

PRIOR-ART-DISCLOSED:

U.S. PATENT DOCUMENTS

PAT-NO	ISSUE-DATE	PATENTEE-NAME	US-CL
4751702	June 1988	Beier et al.	714/13
5978791	November 1999	Farber et al.	707/2

OTHER PUBLICATIONS

"Using Write Protected Data Structures To Improve Software Fault Tolerance in Highly Available

Database Management Systems", Mark Sullivan et al, Proceedings of the International Conference On Very Large Data Bases, pp. 171-179, 1991.

ART-UNIT: 2171

PRIMARY-EXAMINER: Black; Thomas

ASSISTANT-EXAMINER: Chen; Te Yu

ATTY-AGENT-FIRM: Morgan & Finnegan, LLP

ABSTRACT:

A method of detecting and recovering from data corruption of a database is characterized by the step of protecting data of the database with codewords, one codeword for each region of the database; and verifying that a codeword matches associated data before the data is read from the database to prevent transaction-carried corruption. A deferred maintenance scheme is recommended for the codewords protecting the database such that the method of detecting and recovering from data corruption of a database may comprise the steps of protecting data of the database with codewords, one codeword for each region of the database; and asynchronously maintaining the codewords to improve concurrency of the database. Moreover, the database may be audited by using the codewords and noting them in a table and protecting regions of the database with latches. Once codeword values are computed and checked against noted values in memory, a flush can cause codewords from outstanding log records to be applied to the stored codeword table.

26 Claims, 8 Drawing figures

☐ 8. Document ID: US 6151607 A

L14: Entry 8 of 16 File: USPT Nov 21, 2000

US-PAT-NO: 6151607
DOCUMENT-IDENTIFIER: US 6151607 A
**** See image for Certificate of Correction ****

TITLE: Database computer system with application recovery and dependency handling write cache

DATE-ISSUED: November 21, 2000

INVENTOR-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY
Lomet; David B.	Redmond	WA		

ASSIGNEE-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY	TYPE CODE
Microsoft Corporation	Redmond	WA			02

APPL-NO: 09/ 301516 [PALM]
DATE FILED: April 28, 1999

PARENT-CASE:
RELATED APPLICATIONS This is a continuation of U.S. patent application Ser. No. 08/832,870, filed Apr. 4, 1997, which is now U.S. Pat. No. 6,067,550. This is a continuation-in-part of U.S. patent application Ser. No. 08/813,982 now U.S. Pat. No. 5,870,763 and U.S. patent

application Ser. No. 08/814,808 now U.S. Pat. No. 5,946,688, which were both filed Mar. 10, 1997 in the name of David B. Lomet, and are both assigned to Microsoft Corporation.

INT-CL: [07] G06 F 17/00

US-CL-ISSUED: 707/202; 201/205, 201/206, 201/3
US-CL-CURRENT: 707/202; 707/201, 707/205, 707/206, 707/3

FIELD-OF-SEARCH: 707/1-206, 712/209

PRIOR-ART-DISCLOSED:

U.S. PATENT DOCUMENTS

PAT-NO	ISSUE-DATE	PATENTEE-NAME	US-CL
4498145	February 1985	Baker et al.	707/202
5257369	October 1993	Skeen et al.	395/650
5287501	February 1994	Lomet	395/600
5325528	June 1994	Klein	395/650
5371889	December 1994	Klein	395/650
5701480	December 1997	Raz	395/671
5778388	July 1998	Kawamura et al.	707/203
5819304	October 1998	Nilsen et al.	711/5
5826089	October 1998	Ireton	712/209
5832508	November 1998	Sherman et al.	707/200
5857207	January 1999	Lo et al.	707/203

OTHER PUBLICATIONS

Martyn, T., "Implementation design for databases: the forgotten step", IT Professional, Mar.-Apr. 2000, pp. 42-49.
Chang et al., "Deformed shape retrieval based Markov model", Electronics Letters, Jan. 20, 2000, pp. 126-127.
Copeland et al., "Which Web development tool is right for you?", Mar.-Apr. 2000, pp. 20-27.
C. Mohan et al., "ARIES:A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging" ACM Trans. On Database Systems 17,1 (Mar. 1992) 94-162.

ART-UNIT: 271

PRIMARY-EXAMINER: Black; Thomas G.

ASSISTANT-EXAMINER: Jung; David

ATTY-AGENT-FIRM: Lee & Hayes, PLLC

ABSTRACT:

This invention concerns a database computer system and method for making applications recoverable from system crashes. The application state (i.e., address space) is treated as a single object which can be atomically flushed in a manner akin to flushing individual pages in database recovery techniques. To enable this monolithic treatment of the application, executions performed by the application are mapped to logical loggable operations which can be posted to the stable log. Any modifications to the application state are accumulated and the application state is periodically flushed to stable storage using an atomic procedure. The application recovery integrates with database recovery, and effectively eliminates or at least substantially reduces the need for check pointing applications. In addition, optimization

techniques are described to make the read, write, and recovery phases more efficient.

6 Claims, 27 Drawing figures

☐ 9. Document ID: US 6067550 A

L14: Entry 9 of 16 File: USPT May 23, 2000

US-PAT-NO: 6067550
DOCUMENT-IDENTIFIER: US 6067550 A

TITLE: Database computer system with application recovery and dependency handling write cache

DATE-ISSUED: May 23, 2000

INVENTOR-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY
Lomet; David B.	Redmond	WA		

ASSIGNEE-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY	TYPE CODE
Microsoft Corporation	Redmond	WA			02

APPL-NO: 08/ 832870 [PALM]
DATE FILED: April 4, 1997

PARENT-CASE:
RELATED APPLICATIONS This is a continuation-in-part of U.S. patent application Ser. No. 08/813,982 and U.S. patent application Ser. No. 08/814,808, which were both filed Mar. 10, 1997 in the name of David B. Lomet, and are both assigned to Microsoft Corporation.

INT-CL: [07] G06 F 17/00

US-CL-ISSUED: 707/202; 707/203, 707/204, 707/101
US-CL-CURRENT: 707/202; 707/101, 707/203, 707/204

FIELD-OF-SEARCH: 707/1-206

PRIOR-ART-DISCLOSED:

U.S. PATENT DOCUMENTS

PAT-NO	ISSUE-DATE	PATENTEE-NAME	US-CL
5819304	October 1998	Nilsen et al.	711/5
5832508	November 1998	Sherman et al.	707/200
5857207	January 1999	Lo et al.	707/203

OTHER PUBLICATIONS

C. Mohan et al., "ARIES:A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging," ACM Trans. On Database Systems 17 ,1 (Mar. 1992) 94-162.

ART-UNIT: 271

PRIMARY-EXAMINER: Black; Thomas G.

ASSISTANT-EXAMINER: Jung; David Yiuk

ATTY-AGENT-FIRM: Lee & Hayes, PLLC

ABSTRACT:

This invention concerns a database computer system and method for making applications recoverable from system crashes. The application state (i.e., address space) is treated as a single object which can be atomically flushed in a manner akin to flushing individual pages in database recovery techniques. To enable this monolithic treatment of the application, executions performed by the application are mapped to logical loggable operations which can be posted to the stable log. Any modifications to the application state are accumulated and the application state is periodically flushed to stable storage using an atomic procedure. The application recovery integrates with database recovery, and effectively eliminates or at least substantially reduces the need for check pointing applications. In addition, optimization techniques are described to make the read, write, and recovery phases more efficient.

46 Claims, 27 Drawing figures

☐ 10. Document ID: US 5966706 A

L14: Entry 10 of 16 File: USPT Oct 12, 1999

US-PAT-NO: 5966706
DOCUMENT-IDENTIFIER: US 5966706 A

TITLE: Local logging in a distributed database management computer system

DATE-ISSUED: October 12, 1999

INVENTOR-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY
Biliris; Alexandros	Chatham	NJ		
Jagadish; Hosagrahar Visvesvaraya	Berkeley Heights	NJ		
Panagos; Euthimios	New Providence	NJ		
Rastogi; Rajeev R.	New Providence	NJ		

ASSIGNEE-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY	TYPE CODE
AT&T Corp	New York	NY			02
Lucent Technologies Inc.	Murray Hill	NJ			02

APPL-NO: 08/ 803042 [PALM]
DATE FILED: February 19, 1997

INT-CL: [06] G06 F 17/30

US-CL-ISSUED: 707/10; 707/1, 707/8, 707/100, 707/102, 707/2
US-CL-CURRENT: 707/10; 707/1, 707/100, 707/102, 707/2, 707/8

FIELD-OF-SEARCH: 707/3, 707/10, 707/101, 707/4, 707/2, 707/100, 707/102, 707/103, 707/200

PRIOR-ART-DISCLOSED:

U.S. PATENT DOCUMENTS

PAT-NO	ISSUE-DATE	PATENTEE-NAME	US-CL
5261089	November 1993	Coleman et al.	707/1
5410684	April 1995	Ainsworth et al.	395/575
5446884	August 1995	Schwendemann et al.	707/1
5448727	September 1995	Annevelink	707/101
5454102	September 1995	Tang et al.	707/3
5530802	June 1996	Fuchs et al.	395/183.1
5687363	November 1997	Oulid-Aissa et al.	707/4
5721909	February 1998	Oulid-Aissa et al.	707/10
5764877	June 1998	Lomet et al.	707/3
5764897	June 1998	Khalidi	395/200.31
5794229	August 1998	French et al.	707/2
5806065	September 1998	Lomet	707/8

ART-UNIT: 277

PRIMARY-EXAMINER: Amsbury; Wayne

ASSISTANT-EXAMINER: Alam; Shahid

ABSTRACT:

A distributed database management computer system includes a plurality of nodes and a plurality of database pages. When a first node in the computer system updates a first database page, the first node generates a log record. The first node determines whether it manages the first database page. If the first node determines that it manages the first database page, the first node writes the log record to a log storage local to the first node. However, if the first node determines that it does not manage the first database page, the first node then determines whether it includes a local log storage. If the first node includes a local log storage, the first node writes the log record to the local log storage, even if the first node does not manage the first database page. If the first node does not include a local log storage, the first node sends the log record to a second node managing the first database page.

8 Claims, 3 Drawing figures

☐ 11. Document ID: US 5946698 A

L14: Entry 11 of 16 File: USPT Aug 31, 1999

US-PAT-NO: 5946698
DOCUMENT-IDENTIFIER: US 5946698 A

TITLE: Database computer system with application recovery

DATE-ISSUED: August 31, 1999

INVENTOR-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY
Lomet; David B.	Redmond	WA		

ASSIGNEE-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY	TYPE CODE
Microsoft Corporation	Redmond	WA			02

APPL-NO: 08/ 814808 [PALM]
DATE FILED: March 10, 1997

INT-CL: [06] G06 F 17/30

US-CL-ISSUED: 707/202; 395/671, 395/676
US-CL-CURRENT: 707/202; 718/101, 718/106

FIELD-OF-SEARCH: 707/1-10, 707/100-104, 707/200-206, 705/35, 395/671, 395/676, 395/680, 395/200.45, 395/182.17, 364/280

PRIOR-ART-DISCLOSED:

U.S. PATENT DOCUMENTS

PAT-NO	ISSUE-DATE	PATENTEE-NAME	US-CL
5257369	October 1993	Skeen et al.	395/680
5325528	June 1994	Klein	395/680
5371889	December 1994	Klein	395/676
5701480	December 1997	Raz	395/671

OTHER PUBLICATIONS

C. Mohan et al., "Aries:A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging" ACM Trans. On Database Systems 17,1 (Mar. 1992) 94-162.

ART-UNIT: 271

PRIMARY-EXAMINER: Ho; Ruay Lian

ATTY-AGENT-FIRM: Lee & Hayes, PLLC

ABSTRACT:

This invention concerns a database computer system and method for making applications recoverable from system crashes. The application state (i.e., address space) is treated as a single object which can be atomically flushed in a manner akin to flushing individual pages in database recovery techniques. To enable this monolithic treatment of the application, executions performed by the application are mapped to logical loggable operations which can be posted to the stable log. Any modifications to the application state are accumulated and the application state is flushed from time to time to stable storage using an atomic procedure. Applications are recovered by replaying the logged state transition operations, in the same manner that most database systems replay state transformation operations to recover database pages. This application recovery integrates with database recovery, and effectively eliminates or at least substantially reduces the need for check pointing applications. In addition, optimization techniques are described to make the read, write, and recovery phases more

efficient.

39 Claims, 26 Drawing figures

☐ 12. Document ID: US 5933838 A

L14: Entry 12 of 16 File: USPT Aug 3, 1999

US-PAT-NO: 5933838
DOCUMENT-IDENTIFIER: US 5933838 A

TITLE: Database computer system with application recovery and recovery log sequence numbers to optimize recovery

DATE-ISSUED: August 3, 1999

INVENTOR-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY
Lomet; David B.	Redmond	WA		

ASSIGNEE-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY	TYPE CODE
Microsoft Corporation	Redmond	WA			02

APPL-NO: 08/ 826610 [PALM]
DATE FILED: April 4, 1997

PARENT-CASE:
RELATED APPLICATIONS This is a continuation-in-part of U.S. patent application Ser. No. 08/813,982 and now U.S. Pat. No. 5,870,763 U.S. patent application Ser. No. 08/814,808, now pending, which were both filed Mar. 10, 1997 in the name of David B. Lomet, and are both assigned to Microsoft Corporation.

INT-CL: [06] G06 F 17/30

US-CL-ISSUED: 707/202; 707/8, 707/204
US-CL-CURRENT: 707/202; 707/204, 707/8

FIELD-OF-SEARCH: 707/202, 707/8, 707/204

PRIOR-ART-DISCLOSED:

U.S. PATENT DOCUMENTS

PAT-NO	ISSUE-DATE	PATENTEE-NAME	US-CL
4665520	May 1987	Strom et al.	395/182.13
5043866	August 1991	Myre, Jr. et al.	707/202
5222217	June 1993	Blount et al.	707/204
5278982	January 1994	Daniels et al.	707/202
5280611	January 1994	Mohan et al.	707/8
5287501	February 1994	Lomet	707/202

5369757	November 1994	Spiro et al.	395/182.17
5485608	January 1996	Lomet et al.	707/202
5524205	June 1996	Lomet et al.	395/182.14
5581750	December 1996	Haderle et al.	707/202
5594863	January 1997	Stiles	395/182.13

OTHER PUBLICATIONS

C. Mohan et al., "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging" ACM Trans. On Database Systems 17,1 (Mar. 1992) 94-162.
The Recovery Manager Of The Advanced Information Management Prototype, Kuspert et al., IBM Heidelberg Scientific Center, Germany, 1989.
Memory Management And Rollback Recovery In Parallel Architectures, Kun-Lung Wu, University of Illinois at Urbana-Champaign, 1990 (Q.004TA90WU .C001).
Distributed Multi-Level Recovery in Main-Memory Databases Bohannon et al., Bell Labs, Murray Hill, NJ IEEE 1996.

ART-UNIT: 276

PRIMARY-EXAMINER: Amsbury; Wayne

ASSISTANT-EXAMINER: Alam; Shahid

ATTY-AGENT-FIRM: Lee & Hayes, PLLC

ABSTRACT:

This invention concerns a database computer system and method for making applications recoverable from system crashes. The application state (i.e., address space) is treated as a single object which can be atomically flushed in a manner akin to flushing individual pages in database recovery techniques. To enable this monolithic treatment of the application, executions performed by the application are mapped to logical loggable operations which can be posted to the stable log. Any modifications to the application state are accumulated and the application state is periodically flushed to stable storage using an atomic procedure. The application recovery integrates with database recovery, and effectively eliminates or at least substantially reduces the need for check pointing applications. In addition, optimization techniques are described to make the read, write, and recovery phases more efficient.

27 Claims, 27 Drawing figures

☐ 13. Document ID: US 5870763 A

L14: Entry 13 of 16 File: USPT Feb 9, 1999

US-PAT-NO: 5870763
DOCUMENT-IDENTIFIER: US 5870763 A

TITLE: Database computer system with application recovery and dependency handling read cache

DATE-ISSUED: February 9, 1999

INVENTOR-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY
Lomet; David B.	Redmond	WA		

ASSIGNEE-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY	TYPE CODE
Microsoft Corporation	Redmond	WA			02

APPL-NO: 08/ 813982 [PALM]
DATE FILED: March 10, 1997

INT-CL: [06] G06 F 17/30

US-CL-ISSUED: 707/202; 707/200, 711/133, 711/135, 395/676, 395/683
US-CL-CURRENT: 707/202; 707/200, 711/133, 711/135, 718/106, 719/320

FIELD-OF-SEARCH: 707/202, 707/200, 711/133, 711/135, 395/676, 395/683

PRIOR-ART-DISCLOSED:

U.S. PATENT DOCUMENTS

PAT-NO	ISSUE-DATE	PATENTEE-NAME	US-CL
5287501	February 1994	Lomet	707/202
5317731	May 1994	Dias et al.	707/202
5412801	May 1995	De Remer et al.	395/182.18
5485608	January 1996	Lomet et al.	707/202
5524205	June 1996	Lomet et al.	395/182.14
5594863	January 1997	Stiles	395/182.13

OTHER PUBLICATIONS

C. Mohan et al., "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging" ACM Trans. On Database Systems 17, 1 (Mar. 1992) 94-162.

ART-UNIT: 276

PRIMARY-EXAMINER: Black; Thomas G.

ASSISTANT-EXAMINER: Alam; Shahid

ATTY-AGENT-FIRM: Lee & Hayes, PLLC

ABSTRACT:

This invention concerns a database computer system and method for making applications recoverable from system crashes. The application state (i.e., address space) is treated as a single object which can be atomically flushed in a manner akin to flushing individual pages in database recovery techniques. To enable this monolithic treatment of the application, executions performed by the application are mapped to logical loggable operations which can be posted to the stable log. Any modifications to the application state are accumulated and the application state is flushed from time to time to stable storage using an atomic procedure. Applications are recovered by replaying the logged state transition operations, in the same manner that most database systems replay state transformation operations to recover database pages. This application recovery integrates with database recovery, and effectively eliminates or at least substantially reduces the need for check pointing applications. In addition, optimization techniques are described to make the read, write, and recovery phases more efficient.

59 Claims, 26 Drawing figures

☐ 14. Document ID: US 5524205 A

L14: Entry 14 of 16 File: USPT Jun 4, 1996

US-PAT-NO: 5524205
DOCUMENT-IDENTIFIER: US 5524205 A

TITLE: Methods and apparatus for optimizing undo log usage

DATE-ISSUED: June 4, 1996

INVENTOR-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY
Lomet; David B.	Westford	MA		
Spiro; Peter M.	Nashua	NH		
Joshi; Ashok M.	Nashua	NH		
Raghavan; Ananth	Nashua	NH		
Rengarajan; Tirumanjanam K.	Nashua	NH		

ASSIGNEE-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY	TYPE CODE
Oracle Corporation	Redwood Shores	CA			02

APPL-NO: 08/ 050747 [PALM]
DATE FILED: April 21, 1993

PARENT-CASE:
This application is a continuation of, Ser. No. 07/546,306, filed Jul. 2, 1990, abandoned, which is a continuation in part of 07/548,720, filed in Jun. 29, 1990, abandoned.

INT-CL: [06] G06 F 12/16

US-CL-ISSUED: 395/182.14; 395/488
US-CL-CURRENT: 714/16; 711/161

FIELD-OF-SEARCH: 395/575, 395/182.14, 371/12

PRIOR-ART-DISCLOSED:

U.S. PATENT DOCUMENTS

PAT-NO	ISSUE-DATE	PATENTEE-NAME	US-CL
4498145	February 1985	Baker et al.	395/600
4507751	March 1985	Gawlick et al.	395/575
4686620	August 1987	Ng	395/600
4751702	June 1988	Beier et al.	371/9.1
4853843	August 1989	Ecklund	395/600
4868744	September 1989	Reinsch et al.	364/280.3
4878167	October 1989	Kapulka et al.	395/575

4945474	July 1990	Elliott et al.	395/575
5043866	August 1991	Myre, Jr. et al.	395/600

OTHER PUBLICATIONS

VAX Rdb/VMS V4.0 Support Internals Student Guide (Oct. 1, 1990).
Rothermel, et al., "ARIES/NT: A Recovery Method Based on Write-Ahead Logging for Nested Transactions," Proceedings of the Fifteenth International Conference on Very Large Data Bases, (1989), pp. 337-346.
C. Mohan, et al., "A Case Study of Problems of Migrating to Distributed Computing: Data Base Recovery Using Multiple Logs in the Shared Disks Environment," Research Report, (1990), pp. 1-15.
C. Mohan, et al., "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging," Research Report, (1989), pp. 1-45.
Crus, "Data Recovery in IBM Database 2," IBM Systems Journal, vol. 23, No. 2, 1984, pp. 178-188.
Gray, "Notes on Data Base Operating Systems," Research Report, (1978), pp. 1-111.
Kronenberg et al., "VAXclusters: A Closely-Coupled Distributed System," ACM Transactions on Computer Systems, vol. 4, No. 2, May 1986, pp. 130-146.
Lindsay, et al., "Notes on Distributed Databases," Research Report, (1979), pp. 1-57.
Rengarajan et al., "High Availability Mechanisms of VAX DBMS Software," Digital Technical Journal, No. 8, Feb. 1989, pp. 88-98.
C. Mohan, et al., "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging," Research Report, (1989), pp. 1-45.
Stonebraker, "The Case for Shared Nothing," University of California, Berkely, pp. 4-8.
Shoens, "Data Sharing vs. Partitioning for Capacity and Availability," IBM San Jose Research Laboratory K55/281, pp. 10-16.
Shoens, et al., "The Amoeba Project," IEEE (1985), pp. 102-105 Strickland, et al., IMS/VS: An evolving system, IBM SYST J, vol. 21, No. 4, (1982), pp. 490-513.
Bhide, "An Analysis of Three Transaction Processing Architectures," Proceedings of the 14th VLDB Conference, Los Angeles, California, (1988), pp. 339-350.
Kohler, W., "Overview of Synchronization and Recovery Problems in Distributed Databases", Fall COMPCON 80, Sep. 23-25, 1980, pp. 433-441.
Yelavich, B., "Customer Information Control System-An evolving system facility", IBM Systems Journal, vol. 24, Nos. 3/4, 1985, pp. 264-278.

ART-UNIT: 236

PRIMARY-EXAMINER: Baker; Stephen M.

ATTY-AGENT-FIRM: Blakely, Sokoloff, Taylor & Zafman

ABSTRACT:

Each node in a data processing system contains at least one undo buffer and one least one redo buffer for insuring that any changes made to a section of a non-volatile storage medium, such as a disk, can be removed, if a transaction has not been committed, or can be recreated if the transaction has not been committed. The undo buffers each correspond to a different uncommitted transaction. The redo buffer contains the changes made to a copy of the section which is maintained in the memory.

7 Claims, 18 Drawing figures

☐ 15. Document ID: US 5485608 A

L14: Entry 15 of 16 File: USPT Jan 16, 1996

US-PAT-NO: 5485608
DOCUMENT-IDENTIFIER: US 5485608 A

TITLE: Methods and apparatus for updating information in a computer system using logs and state identifiers

DATE-ISSUED: January 16, 1996

INVENTOR-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY
Lomet; David B.	Westford	MA		
Spiro; Peter M.	Nashua	NH		
Joshi; Ashok M.	Nashua	NH		
Raghavan; Ananth	Nashua	NH		
Rangarajan; Tirumanjanam K.	Nashua	NH		

ASSIGNEE-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY	TYPE CODE
Oracle Corporation	Redwood Shores	CA			02

APPL-NO: 08/ 227491 [PALM]
DATE FILED: April 14, 1994

PARENT-CASE:
This application is a continuation of patent application Ser. No. 07/546,454 filed Jul. 2, 1990, now abandoned, which is a continuation in part of Ser. No. 07/549,183 filed Jun. 29, 1990, abandoned.

INT-CL: [06] G06 F 17/30

US-CL-ISSUED: 395/600; 395/182.18, 364/DIG.1, 364/265, 364/268.9, 364/282.1, 364/282.4
US-CL-CURRENT: 707/202; 707/204, 714/20

FIELD-OF-SEARCH: 395/600, 395/575, 395/650, 395/700

PRIOR-ART-DISCLOSED:

U.S. PATENT DOCUMENTS

PAT-NO	ISSUE-DATE	PATENTEE-NAME	US-CL
4189781	February 1980	Douglas	395/575
4498145	February 1985	Baker et al.	395/600
4507751	March 1985	Gawlick et al.	395/575
4686620	August 1987	Ng	395/600
4751702	June 1988	Beier et al.	371/9.1
4853843	August 1989	Ecklund	395/600
4868744	September 1989	Reinsch et al.	395/575
4945474	July 1990	Elliott et al.	395/575
5043866	August 1991	Myre, Jr. et al.	395/600
5062045	October 1991	Janis et al.	395/600
5159669	October 1992	Trigg et al.	395/600
5170480	December 1992	Mohan et al.	395/600

FOREIGN PATENT DOCUMENTS

FOREIGN-PAT-NO	PUBN-DATE	COUNTRY	US-CL
0250847	January 1988	EP	
0295424	December 1988	EP	
2201207	August 1988	GB	

OTHER PUBLICATIONS

Gray, "Notes on Database Operating Systems," 1977, pp. 460-465.
Rothermel, et al., "ARIES/NT: A Recovery Method Based on Write-Ahead Logging for Nested Transactions," Proceedings of the Fifteenth International Conference on Very Large Data Bases, (1989), pp. 337-346.
C. Mohan, et al., "A Case Study of Problems of Migrating to Distributed Computing: Data Base Recovery Using Multiple Logs in the Shared Disks Environment," Research Report, (1990), pp. 1-15.
C. Mohan, et al., "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging," Research Report, (1989), pp. 1-45.
Crus, "Data Recovery in IBM Database 2," IBM Systems Journal, vol. 23, No. 2, 1984, pp. 178-188.
Gray, "Notes on Data Base Operating Systems," Research Report, (1978), pp. 1-111.
Kronenberg et al., "VAXclusters: A Closely-Coupled Distributed System," ACM Transactions on Computer Systems, vol. 4, No. 2, May 1986, pp. 130-146.
Lindsay, et al., "Notes on Distributed Databases," Research Report, (1979), pp. 1-57.
Rengarajan et al., "High Availability Mechanisms of VAX DBMS Software," Digital Technical Journal, No. 8, Feb. 1989, pp. 88-98.
Stonebraker, "The Case for Shared Nothing," University of California, Berkely, pp. 4-8.
Shoens, "Data Sharing vs. Partitioning for Capacity and Availability," IBM San Jose Research Laboratory K55/281, pp. 10-16.
Shoens, et al., "The Amoeba Project," IEEE (1985), pp. 102-105.
Strickland, et al., "IMS/VS: An evolving system," IBM SYST J, vol. 21, No. 4, (1982), pp. 490-513.
Bhide, "An Analysis of Three Transaction Processing Architectures," Proceedings of the 14th VLDB Conference, Los Angeles, Calif., (1988), pp. 339-350.
Walter H. Kohler, "Overview of Synchronization and Recovery Problems in Distributed Databases", IEEE Proceedings on Distributed Computing, Computer Society International Conference, Sep. 23-25, 1980, pp. 433-441.
R. A. Crus, "Data Recovery in IBM Database 2", IBM Systems Journal, vol. 23, No. 2, 1984, pp. 178-188.
A. Yamashita, "Data Base Integrity At Emergency Restart In Data Sharing"; IBM Technical Disclosure Bulletin, vol. 26, No. 2, Jul. 1983, Armonk, N.Y., USA, p. 863.

ART-UNIT: 237

PRIMARY-EXAMINER: Black; Thomas G.

ASSISTANT-EXAMINER: Loomis; John C.

ATTY-AGENT-FIRM: Blakely, Sokoloff, Taylor & Zafman

ABSTRACT:

A data processing system maintains logs for system and media recovery. The logs contain state identifiers each uniquely identifying the state of a corresponding section of a storage medium, such as a disk. The state identifiers are assigned after changes have been made such that they can be determined from the information in the logs. One implementation involves assigning state identifiers in a known sequence, such as in a monotonically increasing sequence of integers beginning with zero.

28 Claims, 18 Drawing figures

☐ 16. Document ID: US 5465328 A

L14: Entry 16 of 16 File: USPT Nov 7, 1995

US-PAT-NO: 5465328
DOCUMENT-IDENTIFIER: US 5465328 A

TITLE: Fault-tolerant transaction-oriented data processing

DATE-ISSUED: November 7, 1995

INVENTOR-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY
Dievendorff; Richard	San Jose	CA		
Mohan; Chandrasekaran	San Jose	CA		

ASSIGNEE-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY	TYPE CODE
International Business Machines Corporation	Armonk	NY			02

APPL-NO: 08/ 181521 [PALM]
DATE FILED: January 14, 1994

FOREIGN-APPL-PRIORITY-DATA:

COUNTRY	APPL-NO	APPL-DATE
GB	9306649	March 30, 1993

INT-CL: [06] G06 F 11/00

US-CL-ISSUED: 395/182.13; 395/600, 395/650
US-CL-CURRENT: 714/15; 707/202

FIELD-OF-SEARCH: 395/575, 395/600, 395/650, 371/12, 364/282.4, 364/282.1, 364/281.3

PRIOR-ART-DISCLOSED:

U.S. PATENT DOCUMENTS

PAT-NO	ISSUE-DATE	PATENTEE-NAME	US-CL
4945474	July 1990	Elliott	364/200
5258982	November 1993	Britton et al.	370/110.1
5261089	November 1993	Coleman et al.	395/600
5319773	June 1994	Britton et al.	395/575
5333303	July 1994	Mohan	395/575

ART-UNIT: 243

PRIMARY-EXAMINER: Beausoliel, Jr.; Robert W.

ASSISTANT-EXAMINER: Snyder; Glenn

ATTY-AGENT-FIRM: Keohane; Stephen T.

ABSTRACT:

In transaction processing systems, it is known for resource-updating operations within a transaction to be backed out at the request of an application program following detection of error conditions during processing of the transaction. If the error condition is very likely to recur, it may be undesirable for the operations request to be presented to the application exactly as before. A transaction-oriented data processing system and a method of transaction-oriented data processing are provided in which operation requests or data packets may be marked to be excluded from the effects of application-requested backouts.

10 Claims, 5 Drawing figures

Term	Documents
13.PGPB,USPT,USOC,EPAB,JPAB,DWPI,TDBD.	16
(L13).PGPB,USPT,USOC,EPAB,JPAB,DWPI,TDBD.	16